4

Technical Report 1205

Automated Acquisition of Evolving Informal Descriptions

Howard B. Reubenstein

MIT Artificial Intelligence Laboratory

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 1205	2. GOVT ACCESSION NO	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Automated Acquisition of Evolving Informal Descriptions		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Howard B. Reubenstein		8. CONTRACT OR GRANT NUMBER(s) IRI-8616644 N00014-88-K-0487 CCR-898273
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE June 1990
		13. NUMBER OF PAGES 227
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Knowledge acquisition	Reuse
Requirements Analysis	Cliche-based reasoning
Informality resolution	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Much of human communication proceeds via an exchange of informal descriptions characterized by *ambiguity*, *contradiction*, and *incompleteness*. This thesis describes an automated system, the Listener, that unintrusively performs *knowledge acquisition* from informal input. The Listener develops a coherent internal representation of a description from an initial set of disorganized imprecise statements. Each statement

(continued on back)

Block 20 continued:

provides fragmentary, ambiguous, possibly inconsistent information. The description and its implications are checked for consistency, completeness, and conclusions deemed interesting. The Listener interactively presents the results of these checks as a guide to continuing the acquisition process. The Listener also produces a summary document from its internal representation in order to facilitate communication, review, and validation. These behaviors are supported by a variety of techniques, including dependency-directed reasoning, hybrid knowledge representation, and the reuse of common forms (*clichés*).

The Listener system is an effective knowledge acquisition tool in domains where: the initial description of a problem or idea is vague and there exists a domain model that provides both common terminology with which to communicate and expectations regarding an evolving description. To demonstrate the breadth of the Listener's capabilities, the Listener has been used in the domain of legal documents to produce a consulting agreement and in the software requirements acquisition domain.

The Requirements Apprentice (RA) is the Listener implemented in the domain of software requirements analysis. Requirements acquisition is one of the most important and least supported parts of the software development process. The RA assists a human analyst in the creation and modification of software requirements. Unlike most other requirements analysis tools, which start from a formal description language, the focus of the RA is on the transition between informal and formal specifications. The RA supports the earliest phases of creating a requirement in which informality is an inevitable feature.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. T.R. No. 1205 June 1990

Automated Acquisition of Evolving Informal Descriptions

by
Howard B. Reubenstein

ABSTRACT

Much of human communication proceeds via an exchange of informal descriptions characterized by *ambiguity*, *contradiction*, and *incompleteness*. This thesis describes an automated system, the Listener, that unintrusively performs *knowledge acquisition* from informal input. The Listener develops a coherent internal representation of a description from an initial set of disorganized imprecise statements. Each statement provides fragmentary, ambiguous, possibly inconsistent information. The description and its implications are checked for consistency, completeness, and conclusions deemed interesting. The Listener interactively presents the results of these checks as a guide to continuing the acquisition process. The Listener also produces a summary document from its internal representation in order to facilitate communication, review, and validation. These behaviors are supported by a variety of techniques, including dependency-directed reasoning, hybrid knowledge representation, and the reuse of common forms (*clichés*).

The Listener system is an effective knowledge acquisition tool in domains where: the initial description of a problem or idea is vague and there exists a domain model that provides both common terminology with which to communicate and expectations regarding an evolving description. To demonstrate the breadth of the Listener's capabilities, the Listener has been used in the domain of legal documents to produce a consulting agreement and in the software requirements acquisition domain.

The Requirements Apprentice (RA) is the Listener implemented in the domain of software requirements analysis. Requirements acquisition is one of the most important and least supported parts of the software development process. The RA assists a human analyst in the creation and modification of software requirements. Unlike most other requirements analysis tools, which start from a formal description language, the focus of the RA is on the transition between informal and formal specifications. The RA supports the earliest phases of creating a requirement in which informality is an inevitable feature.

Acknowledgments

I am grateful to Dick Waters, my thesis advisor, for providing just the right combination of freedom, advice, and insight and for helping me learn how to tell the story. Dick worked his way through many revisions of this thesis and helped improve it immeasurably.

Chuck Rich and Yishai Feldman implemented and supported CAKE providing a foundation on which this work was built. Chuck was always available to provide insight on any of a number of topics. His careful reading of this thesis is greatly appreciated.

Peter Szolovits and Ramesh Patil were the other members of my committee. I would like to thank them for their feedback and time.

Paul Lefelhocz implemented the air-traffic control scenario, and as the first user of the RA provided valuable data in support of this thesis.

I'd also like to thank my officemates and other members of the Programmer's Apprentice group for technical and non-technical discussions and general comradery: Yang Meng Tan, Linda Wills, Paul Lefelhocz, Bob Hall, Rudolph Seviora, Ron Kuper, Yishai Feldman, and Dilip Soni.

I'd like to thank Dr. Eric Alani for getting there first. Roy and the whole New Years crew for non-computerized entertainment, support, and friendship.

My parents are always there with love, trust, and support.

Thanks Lori for being my partner in adventure, for laughing and smiling with me. I'm glad I didn't wait.

This report is a revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on May 25, 1990 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

The research described here was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the following organizations: National Science Foundation under grants IRI-8616644 and CCR-898273, Advanced Research Projects Agency of the Department of Defense under Naval Research contract N00014-88-K-0487, IBM Corporation, NYNEX Corporation, and Siemens Corporation.

Contents

Chapter 1

Overview: The Listener

"Just *listen* to yourself" is a common exhortation to someone who is not making sense. Monitoring what someone says for surprising conclusions or blatant contradictions is a good way to debug ideas. This thesis concerns an automated *listener system* that processes an informal description of an idea and creates a coherent formal representation of that idea. This representation can be used both to communicate the idea to other people and as input to further automated processing.

The problem this thesis addresses is at the heart of any process that requires formalization of ideas that initially exist only in peoples' minds. It is unknown whether the mental representation of ideas is formal or coherent, but it is clear that their initial exposition is typically informal and vague. The problem is how to capture a complex idea from a description that is characteristically partial, underspecified, potentially contradictory, and poorly ordered.

Two people, a speaker and a listener, can not communicate efficiently if the speaker is required to give the myriad details required to fully describe the idea to be shared. If a shared view of the world can be accessed by both participants, then the conversation can be shortened. Human speakers use "informal" conversational techniques that access such shared knowledge. In order for this sort of abbreviated communication to succeed, the listener must recover the information implicit in the speaker's utterances. The listener resolves informalities by extracting relevant information from the shared view of the world and incorporating it into the understanding of the description.

The problem of description acquisition is simplified in this thesis by assuming that the speaker maintains the initiative in the conversation. This precludes the more dynamic situation of two equals engaged in a conversation, negotiating positions and achieving a consensus. The listener's function is to understand what the speaker is saying. The listener is unintrusive, holding its questions to a minimum. It assumes the speaker is knowledgeable and will clarify the description as time progresses. However, in order to assist the speaker in debugging his ideas, the listener provides some immediate feedback to indicate an evolving understanding of the description. The listener may ask a few questions, mainly to obtain guidance from the speaker in fixing problems in the description.

A prototype listener system has been built to explore the kind of knowledge acquisition described above. A specialized version, called the Requirements Apprentice (RA), listens to software requirements descriptions. The RA has successfully processed two example informal requirements descriptions producing coherent, formal internal representations of them. The RA demonstrates the feasibility of knowledge acquisition from a particular kind of "noisy" input signal. The listener system has also been applied to a short example in a legal domain as an initial demonstration of the range of the developed techniques.

This thesis presents: a characterization of the kinds of informality that may exist in a description, techniques for resolving such informality, a preliminary theory of knowledge reuse in the domain of software requirements acquisition, and a working prototype that demonstrates these ideas. The exposition in this first chapter abstracts away from the specific application supported by the RA and concentrates on the general techniques used in listening to descriptions. (In describing the domain-independent behavior of the RA, the system will be referred to as the 'Listener'. Where the requirements application matters, the system will be referred to as the 'RA'.)

1.1 Listener Architecture

The basic architecture of the Listener is shown in Figure 1-1. (The italicized words in the following paragraphs refer to labels in the figure.) The input to the system is a series of statements that, taken as a whole, are an *informal input* description of the ideas being explored. These statements are expressed in a command language that provides a "correct parse" of the input and a translation into first order logic. This command language is based on an entity-relationship view of the world. The basic commands permit: declaring the existence of a named object, classifying the object, providing role values, and stating relations and constraints. The speaker must have some familiarity with this command language in order to communicate with the Listener. By assuming a speaker trained in this manner, the Listener can avoid dealing with problems that are better studied in the context of natural language comprehension. Natural language is characterized by additional kinds of informalities, e.g., anaphora, ambiguous phrasal and clausal attachment, lexical category ambiguity, and omitted phrases, that have been studied elsewhere (e.g., [1, 2]).

As each statement of the input description is presented to the Listener, the system provides *immediate feedback* to help show the speaker that the statement has been properly understood. Feedback includes rephrasing of the statement and reporting of certain interesting deductions.

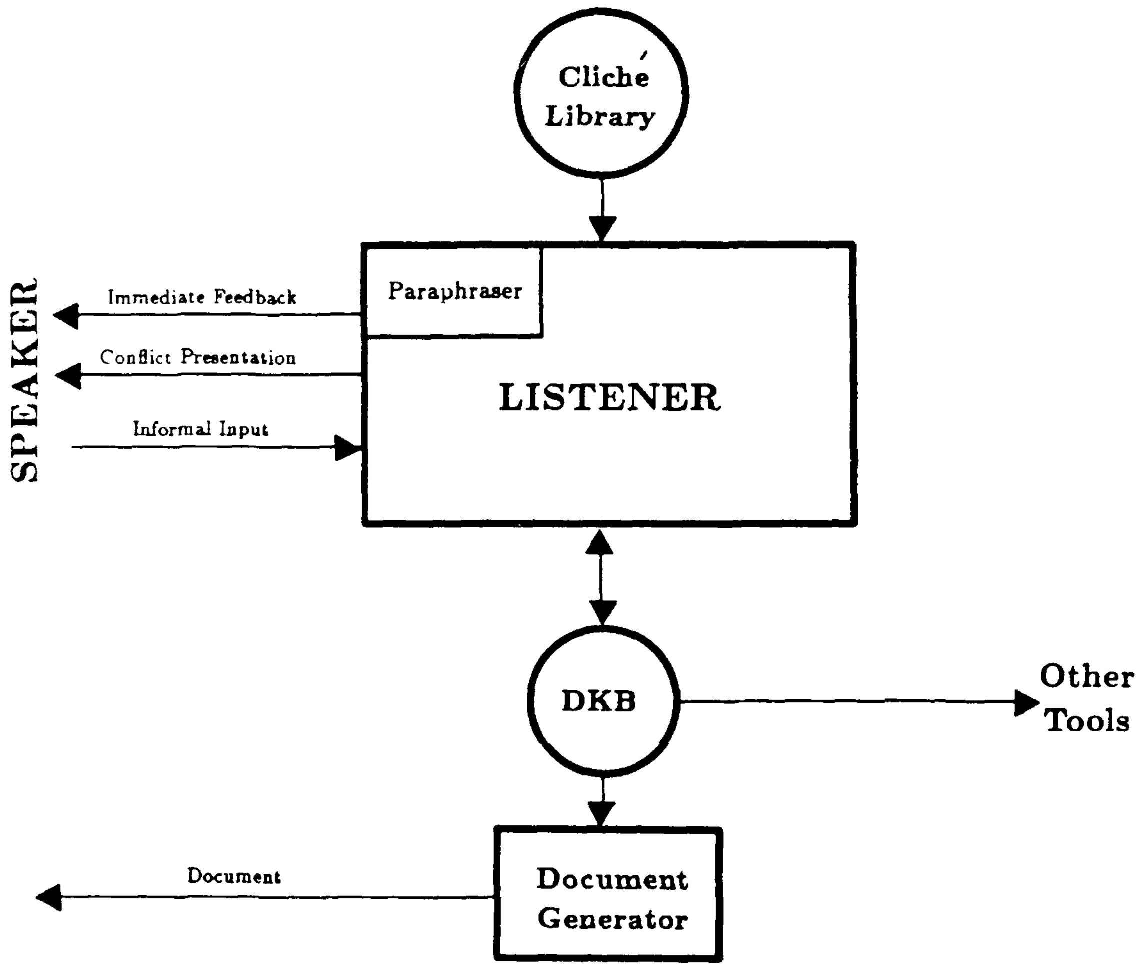

Figure 1-1: Listener Architecture

The Listener helps the speaker debug his ideas by detecting *conflicts* in the evolving description. A conflict is either an outright logical contradiction or an unusual (heuristically defined) combination of facts. When a conflict is detected, the Listener displays the reasoning underlying the problem, including the supporting premises. The Listener then provides an opportunity to alter the description to correct the problem. If the problem is not corrected, the Listener continues to monitor for new information bearing on the problem's resolution.

The *paraphraser* generates the immediate feedback. It consists of two processes. The first process outputs a summary of the command presented. The second process notices new interesting facts and presents them. In fact, the second process may assume the duty of the first one since a paraphrase of the input statement often follows directly from th new implications of the input.

The Listener has access to a library of reusable knowledge chunks called clichés. Understanding is formed in the context of this body of information shared with the speaker. The *cliché library* is used to assist the Listener in understanding a description. It defines the ontology of the ideas being understood and describes basic domain concepts to be used as building blocks in the current acquisition. The speaker refers to clichés both explicitly and implicitly. Understanding a statement often reveals implicit references.

Resolution of informality is crucial to the *Listener's* attempts to understand the input description. Each cliché contains information that may enable the Listener to resolve some informality in the input and thus obtain a better understanding of the idea being described. Each statement of the description, and the processing used to remove informality, results in the classification of various objects as instances of particular clichés. This classification results in information in the cliché definition being propagated to the relevant objects. Thus, these cliché definitions provide information that helps elaborate the evolving representation of the idea.

Contradiction detection is the primary technique for discovering problems in the description. As more and more cliché definitions are propagated, the "density of information" about objects increases. This makes it more and more likely that problems in the description will appear as logical contradictions. Cliché definitions assert information chunks at a time, thus they raise the density of information quicker than individual logical assertions. These definitions limit the subspace of representable ideas to a set of ideas that are well-formed with respect to the structure of the world described in the cliché library.

The formal representation of the idea being acquired is stored in the *description knowledge-base* (DKB). The Listener attempts to create a consistent and detailed network of assertions in the DKB. No guarantees can be made, however, that this network is the correct representation of the speaker's idea. The DKB may also contain explicitly represented, unresolved contradictions. The DKB captures the current state of understanding in a computer-based form for use by other processes. Figure 1-1 indicates that the Listener is meant to be a modular component in a suite of *tools*. The results of t' Listener's work are communicated through the DKB.

The *document generator* is a tool that produces a summarization of the ideas currently represented in the DKB. Document generation supplements the immediate feedback by providing a global view of the evolving description. The document provides a coherent restatement of the idea being acquired. The term 'coherent' implies that the speaker's informalities have been resolved. Production of the document is supported by canned text in the cliché library and a procedurally encoded top-level organization that guarantees access to all data that needs to be reported. The document is read by the speaker for the purpose of validation. If the speaker chooses to modify the description in response to a review of the document, the Listener system processes the changes and the evolution of the idea continues.

User Interface

Before describing the specific capabilities of the Listener, it is useful to focus on the intended interface to the system as sketched in Figure 1-2. (The italicized words in the following paragraphs refer to labels in the figure.) The research prototype has a much simpler interface that permits demonstrating the adequacy of the underlying technology.

The user interface can assist the Listener in achieving the goal of being unintrusive while it proceeds to integrate the input description and information from the cliché library.

The bottom window is for the *dialog*. This is where the speaker presents the input description and the paraphraser presents its rephrasing and deductions.

Next to the dialog is an agenda of *pending issues*. Agenda items include: information the Listener would like to know to deepen its understanding, pending informalities that the Listener is trying to resolve, and low priority messages and observations. As information appears in this window, the speaker can glance at it and either provide the requested information, if prepared to do so, or simply ignore it.

Above the dialog window are alternative presentation windows that provide different viewpoints of the DKB. The *document* window provides a view of the acquisition process as the authoring of a document. The Listener presents portions of this document as they change with each statement of input. This metaphor is somewhat problematical since each input statement can effect many parts of the document. A notion of focus in the document could be introduced to address this. The current implementation does not provide these capabilities nor does the thesis address these problems.

Whenever a conflict is detected, there is a potentially complicated chain of rea-

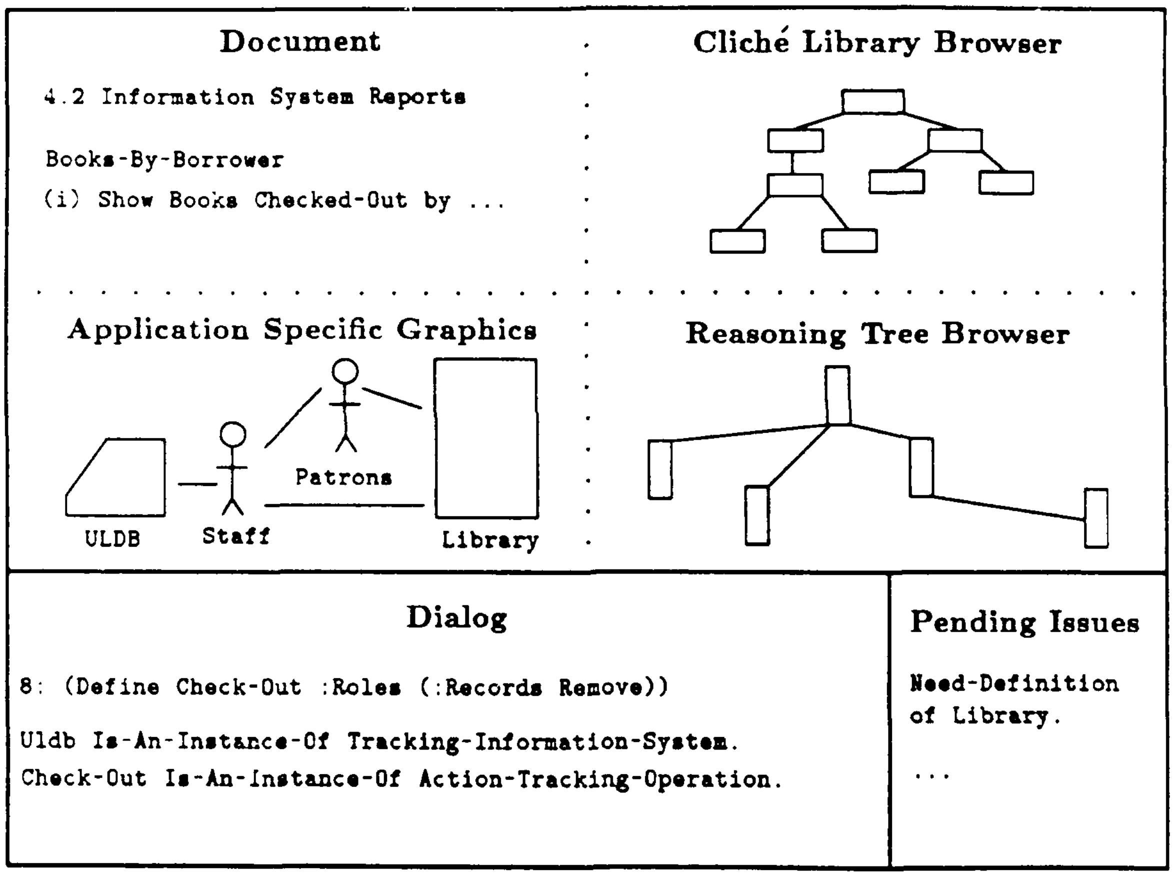
Document
4.2 Information System Reports
Books-By-Borrower
(i) Show Books Checked-Out by ...
Cliché Library Browser
Application Specific Graphics
ULDB
Staff
Patrons
Library
Reasoning Tree Browser
Dialog
8: (Define Check-Out :Roles (:Records Remove))
Uldb Is-An-Instance-Of Tracking-Information-System.
Check-Out Is-An-Instance-Of Action-Tracking-Operation.
Pending Issues
Need-Definition of Library.
...

Figure 1-2: Proposed User Interface

soning behind it. The *reasoning tree browser* is a pop-up window that displays the support tree of a conflict. It also provides for navigation through the explanation to facilitate the resolution of the conflict.

To facilitate access to the shared knowledge, the speaker can use the *cliché library browser* to gain a better idea of what the Listener knows and how to communicate with it.

In the RA, a diagramming window would be useful ***application specific graphics*** to be used as a traditional source of requirements input.

As the capabilities of the Listener are described it may be useful to keep in mind the different sources of information available to the speaker. The current prototype provides all of this information though not in the most convenient form.

1.2 Capabilities

From the speaker's viewpoint, the capabilities of the Listener can be grouped into the following five categories: use of clichés, informality resolution, support for evolution, incremental feedback, and summarization. These are explained in detail in Section 3.1.

Clichés are the heart of the language shared by the speaker and the Listener, providing a common terminology for the transfer of knowledge between them. Clichés are valuable, in part, because they provide an effective way to communicate. Each statement in a description provides a fragmentary addition to an evolving representation of an idea. The idea is built up by creating objects, fleshing out an object's structure through classification and role assignments, and stating constraints that exist between objects. Acquisition of the idea is accelerated when the speaker can refer to components of the cliché library instead of providing that information in detail. Acquisition is also accelerated when the Listener can access the cliché library as a source of expectations.

The notion of informality is discussed in more detail later in the thesis. Briefly, it refers to statements (and descriptions as a whole) that may be: ambiguous, incomplete, inaccurate, and contradictory. In addition, the use of abbreviation and poor ordering of input reflect informality in the description.

Informality is an inevitable property of the speaker's description to the Listener. It is not a matter of the speaker being lazy or incompetent. Informality is an essential part of the human thought process. It is used as part of a powerful debugging strategy that starts with an approximate solution and refines that solution until it is acceptable [3].

Informality hides information from the Listener. To fully understand what the speaker is saying, the Listener must recover this hidden information by resolving the informalities in the speaker's input description. The automated recovery of this implicit information facilitates a more efficient conversation since the speaker is not forced to describe everything in tedious detail. This recovery, however, requires a source of relevant information to assist in the detection and correction of informalities. The cliché library is the source of this additional information.

Support for evolution entails supporting changes in the description. As an acquisition session proceeds, the speaker may change his mind about any part of the description presented thus far. Such changes may be stimulated by feedback provided by the Listener, or the speaker may simply desire to change a previous input. In any case, the idea must be allowed to evolve in whatever direction the speaker wishes to take it. Evolutionary acquisition is characterized by:

- incrementality - descriptions are presented in a piecemeal fashion.
- order independence - the fragments of the description are not presented in any predefined order and the order of presentation does not matter in the representation of the idea ultimately acquired.
- redefinition - the description process is not purely additive. The speaker may alter previous parts of the description.

In order to assist the speaker in debugging the description, the Listener provides feedback regarding the evolving description. Incremental feedback provides an evolving picture of the state of the conversation. Without such feedback, the speaker would have no data on which to base possible corrections to the description being presented. This feedback highlights outright contradictions and new deductions that may not be obvious to the speaker. It is assumed that the speaker will digest the feedback and that this will either: trigger new lines of thought, highlight problems in the current description, or confirm the current line of thought.

The above four capabilities address the needs of a conversational interaction between speaker and listener. Summarization (the generation of a summary document) supports the more contemplative process of an editing cycle between author and reviewer. The document can be read and reviewed in a setting outside of the Listener system and therefore used as a medium for the transfer of information to other people concerned with the acquisition process. This capability is not necessary for statement-by-statement understanding; however, it provides a comprehensive, global view of the results of the description acquisition, complementing incremental feedback.

1.3 Techniques

The following set of tools and techniques are central to providing the Listener with the capabilities described above: CAKE, cliché-frames, classification, plausibility checking, and contradiction processing. The use of these techniques will be demonstrated in the scenarios in the next chapter and then explained in detail in Section 3.2.

CAKE is the reasoning system on top of which the Listener is built. Its features include: a predicate calculus representation, a type lattice, and truth maintenance. In addition to supporting the basics of the knowledge representation and reasoning schemes used in the Listener, CAKE also provides a base for incremental, order invariant deduction. The Listener builds on top of this and structures its reasoning so that these properties are preserved.

Clichés are represented using a frame construct called a cliché-frame. A cliché consists of a set of roles and a set of constraints. Roles are placeholders to be filled by other objects. Constraints relate the roles. The choices made in filling the roles determine a particular instantiation of the cliché. Cliché-frames represent three kinds of information about a cliché:

- Definitional - the role/constraint structure. Definitional information is true of each instance of the cliché-frame.
- Computational - Computational information assists in performing some task. For example, canned text associated with a cliché assists in the production of text to be included in a summary document.
- Predictive - Predictive information provides an index to the facts that will become true if an object is asserted to be an instance of the cliché-frame. It is used to help the Listener resolve informalities that require an abductive choice to be made.

Classification of objects is one process by which the Listener forms an understanding of the idea being described. As the Listener learns more about an object, its classification in the hierarchy defined by the cliché libraries is refined and made more specific. The more specifically an object can be classified, the more information about it that can be propagated. The classification algorithm used is different than the standard KL-ONE-style algorithm [4] in that it makes use of contingent information in its processing.

Plausibility checking is another mechanism used to support informality resolution. It can be divided into two classes of processing: type inference and abductive reasoning. A simple example that captures the flavor of plausibility checking is as follows: if "the height of X is 68 inches," then type inference requires X to be the kind of thing in which it makes sense to talk about X's height, e.g., X is a physical-object. Abductive reasoning might further infer that X is a human, if humans typically have heights in the appropriate range.

Contradiction processing is the primary mechanism used to debug descriptions. CAKE supports contradiction detection. Contradictions can be repaired using domain independent logical techniques or using a set of heuristics that relies on information in the domain model to provide data for alternatives.

1.4 Contributions

This thesis makes contributions in the following areas:

- The utility of informality in description presentation is demonstrated, as is the difficulty of acquiring knowledge from descriptions that incorporate such informalities. The nature of this informality is characterized and a set of techniques to remove informality from descriptions is proposed. The process of removing informality from a description is the main technique used in developing a coherent description.
- A conversational model of knowledge acquisition from a noisy data source is proposed as a way to accelerate and automate what has been, at least in software requirements acquisition, a manual and error prone task.
- Reusable components of software requirements (clichés) are identified and codified. A representation for these components is proposed. This representation implements a preliminary theory of requirements clichés and is used to encode a model of requirements acquisition. A domain model, composed of clichés, is used to set up expectations and provide effective assistance in resolving informality and supporting knowledge acquisition. The utility of investing in the construction of such models is demonstrated.
- The feasibility of automated assistance for software requirements acquisition and analysis is demonstrated by a working prototype listener system called the RA. The RA demonstrates the integration of a system implementation on top of an active knowledge base to produce a knowledge base which captures both the artifact and its state of development. Requirements are identified as an example of a continually evolving artifact, as opposed to a static entity, that requires a dynamic acquisition tool.

This chapter has described a listener that attempts to capture an idea in a coherent internal representation. The power of this listener is provided by a suite of capabilities and functions uniformly applied, acting in synergy towards the same goals.

1.5 Thesis Organization

Chapter 2 discusses a Listener, called the RA, implemented in the software requirements acquisition domain. The problem of requirements acquisition is defined. An annotated running scenario is presented of the RA acquiring a library database requirement. Another scenario involving the requirements for an air-traffic control system is also presented.

Chapter 3 discusses the capabilities that permit the Listener to perform as demonstrated. It also discusses the techniques used to implement them and some of the more interesting implementation details.

Chapter 4 presents a smaller example of the Listener working on a problem in an entirely different domain: the description of a legal contract.

Chapter 5 evaluates the Listener system by considering issues addressed and trade offs made in the construction of the Listener system.

Chapter 6 discusses related work. Chapter 7 discusses future work and conclusions[1].

[1]All footnotes in this thesis, except one, can be skipped over–they simply provide information such as scenario cross-references and pointers to implementation files.

Chapter 2

A Listener for Software Requirements Acquisition

This chapter discusses an application of the Listener system to the software requirements acquisition domain. The reader may want to skip briefly to the scenario in Section 2.8 on page 29 to get a better idea of the precise kind of interaction supported by the RA. The beginning of this chapter defines the requirements acquisition problem.

Requirements acquisition is a multi-faceted problem that requires, among other skills, the ability to engage in skilled listening. The RA is a Listener system that has been configured and augmented to operate in the requirements acquisition domain.

The RA is a component of the Programmer's Apprentice (PA) [5] that assists in the initial stages of software development when the problem statement and needs are vague. The RA's job is to turn an informal requirements description into a formal, computer-based description. The RA produces a requirements document that can be used for communication, review, and validation. This document is produced from a requirements knowledge base that can be used as a medium to support further analysis and design. It is intended that the RA will eventually be hooked up to a design assistant [6] that will help continue the software construction process.

It is important to keep in mind the position occupied by the RA in the requirements acquisition process. As illustrated by figure 2-1, the RA is intended to assist a requirements ***analyst*** not the client or end-users.

The analyst begins the requirements acquisition process by means of a "skull session" that elicits an informal idea of what is wanted. A good example of the result of such a skull session is the library database requirement sketch [7] in figure 2-5 on page 30.

Starting from such an informal requirements sketch, the RA assists an analyst (and through the analyst, the end-user) in arriving at a more complete and more formal

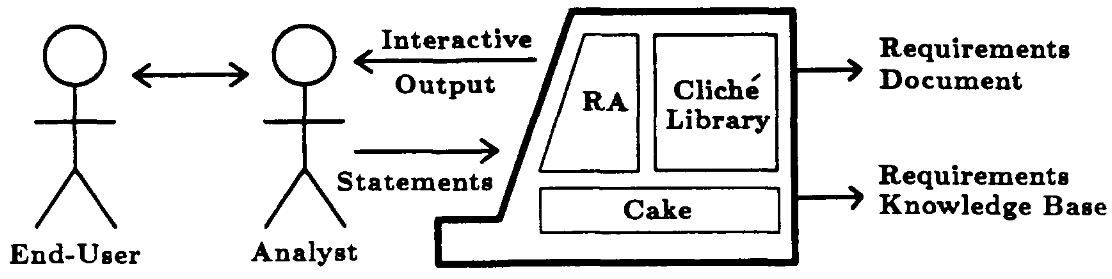

Figure 2-1: The Requirements Apprentice.

requirement. In doing this, the RA continues in the tradition of the SAFE project [1] and differs from most current work on software requirements tools (e.g., [8, 9, 10]) that focuses instead on the validation of a requirement that is already stated in a formal way.

The RA produces three kinds of output. Interactive output notifies the analyst of conclusions drawn and inconsistencies detected while requirements information is being acquired. A machine-manipulable requirements knowledge base (RKB) represents everything the RA knows about an evolving requirement. The RKB will eventually be used to communicate with other tools working on other parts of the software process. Finally, the RA will be able to create various kinds of written documents based on the contents of the RKB. Documents can be produced for use by the end-user (for validation) or for use by a system designer.

2.1 What is a Requirement?

A software requirement is many things. In some cases, its most important role is as a contract between a supplier and a customer. In this guise, it can be viewed as defining an acceptance test for the product delivered by the supplier. In its role as a contract, however, it depends on the reasonableness of the parties involved. This is necessary because a requirement attempts to perform the difficult task of drawing boundaries around the desired system behavior without being too narrow (thus omitting some necessary features), too broad (thus admitting the possibility of irrelevant and costly behaviors), or too specific (thus eliminating effective design and implementation choices).

The simplest functional view of a requirement is that it is a predicate on behavior that can be used to judge whether a system is operating correctly. To serve in this capacity, the requirement must capture all the informational requirements of the system. This requires identifying all inputs and outputs relevant to a situation and dividing the input space into appropriate equivalence classes, each of which has its own particular output specification.

A requirement can also be used simply as a communication device between groups of people who are working towards a common goal. The requirement communicates an evolving sense of the problem and solution between the parties involved. The RA supports this sense of software requirements acquisition. The problem the RA is interested in solving is a communications problem. In this light, the desiderata of a good requirement are:

- readability - the presentation must be clear and predictable. Clarity is improved by good organization, proper cross-referencing and good indexing. Clarity is also improved by consistent use of unambiguous terminology.
- robustness - the description must enumerate requirements for all relevant situations and not omit important details. Details are often implicit in context but need to be made explicit in order to verify their correctness. A description can be incomplete if it touches on a topic but does not include enough relevant details to determine a solution or if it touches on a topic but does not mention closely related considerations.
- implementation unbiased - the requirement should specify functionality in implementation unbiased terminology. Someone with intimate knowledge of an implementation technology may formulate a description of a problem by stating a solution in terms of that implementation technology and requiring that the system perform in that manner. In this case, the person has failed to formulate an abstract description of the problem and further has hidden assumptions in his mapping to implementation technology.

Scope and Limitations of Requirements Modeling

Realistic requirements modeling entails capturing a variety of different kinds of information including: project scheduling constraints, business procedure impact, environment induced hardware and software constraints, performance requirements, and security constraints. An industrial strength tool should make provision for, at the very least, the textual capture of such information. The RA only assists in the acquisition of the basic information requirements of a system, i.e., what needs to be computed from what data.

The basic goal of the RA is to acquire a description of the problem and its solution. It captures this information in a descriptive, as opposed to operational, representation. In particular, the RA does not support a simulation capability. Another limitation of the RA is that it focuses primarily on problems of underspecification. Overspecification, e.g., describing how to implement a function versus describing what the function is to compute, is a potentially serious problem in requirements acquisition. Essentially, the RA avoids such problems by not providing clichés that facilitate discussion of detailed implementation decisions.

2.2 Flaws in Current Requirements

Today, the primary output of a software requirement acquisition process is a requirements document. For large systems, this document may be 8.5 x 11 x 11 [11]. (The second 11 is in *feet* and refers to the height of the stack of paper!) The large size of a requirements document is not in itself a flaw. Some problems are of such inherently great complexity that they require sizable descriptions. However, flaws are often introduced in the creation of the document by manual processes. A set of problems which can exist in requirements documents is enumerated in [12]. The RA addresses many of these as forms of informality to be resolved in the input description. The document produced by the RA is machine generated and can therefore avoid various presentation problems. The "seven deadly sins" listed in [12] are:

- noise (including redundancy) - the document includes irrelevant statements or unnecessary repetition of statements. Not all noise is bad, since there are expositional reasons to repeat descriptions as long as multiple occurrences are kept coordinated.
- incompleteness (silence) - with all the detail present in a huge requirements document, there is still no guarantee of the coverage of the requirement. Relevant descriptions may be underspecified or left out entirely.
- overspecification - is a problem with either having said too much or having phrased the description using an implementation-biased vocabulary.
- contradiction - large documents may contain directly contradictory statements (with some textual separation) or may imply contradictory conclusions.
- ambiguity - can be viewed as underspecification. The ambiguous term or phrase encompasses a number of possibilities from which the meaning of the term needs to be selected.
- forward reference - the order of description in a document is not necessarily designed with readability in mind.

- wishful thinking - is a problem of superficial analysis. The requirement calls for things which upon further, in-depth analysis are seen to be impossible.

These flaws are exacerbated by the fact that a paper requirements document is a static snapshot of an ongoing process of requirements acquisition. It is a snapshot which is expensive to take and thus not taken often. It is a snapshot which is not directly tied to the data of the ongoing process since that data generally exists in sketches and in people's heads.

These problems argue for a new approach to software requirements acquisition. As a straw man, one might have proposed a system that attempts to read and understand a requirements document and then produces a version without the problems listed above. Even if this can be successfully done, however, one has only succeeded in sanitizing the snapshots. Nothing will have been done to support the evolutionary nature of the process of requirements acquisition itself.

2.3 Knowledge Acquisition and Informality

The RA takes a "conversational" approach to the capture of software requirements. The heart of this approach is to accept an evolving informal statement of the requirement and resolve the informalities to produce a formal requirement.

By describing the problem of requirements acquisition in a more precise manner, some insight into the nature of informal and formal descriptions can be obtained. We assume here that the requirement is being acquired from a single person. The RA makes the slightly less restrictive assumption that there exists a single analyst who interacts with the RA and represents the potentially diverse client community. We also assume that this analyst can be modeled as having a representation of the requirement in his head that consists of a logical model and a set of axioms in first order logic (FOL) that captures properties of the model. (For all practical purposes, the RA uses FOL as its basic representation language.)

Ideally, we would like to have some sort of oracle that could extract this model and capture a set of FOL axioms appropriate to the model. We don't. The situation is more like that portrayed in figure 2-2. The idea to be transferred (the cloud in the upper left hand corner) is an isolated artifact, i.e., there is no direct access to it. The analyst cannot convey the exact idea to the RA since the mind is closed to exact probing. Neither can the document produced by the RA be validated for 100% accuracy. The document only captures the idea approximately. Since there is no solid (correctness preserving) path from the idea to an external representation, there is also no means of wholesale extraction of the idea.

The RA supports a formulation of requirements acquisition which assumes the analyst has, at any particular moment, access to FOL statements that partially describe

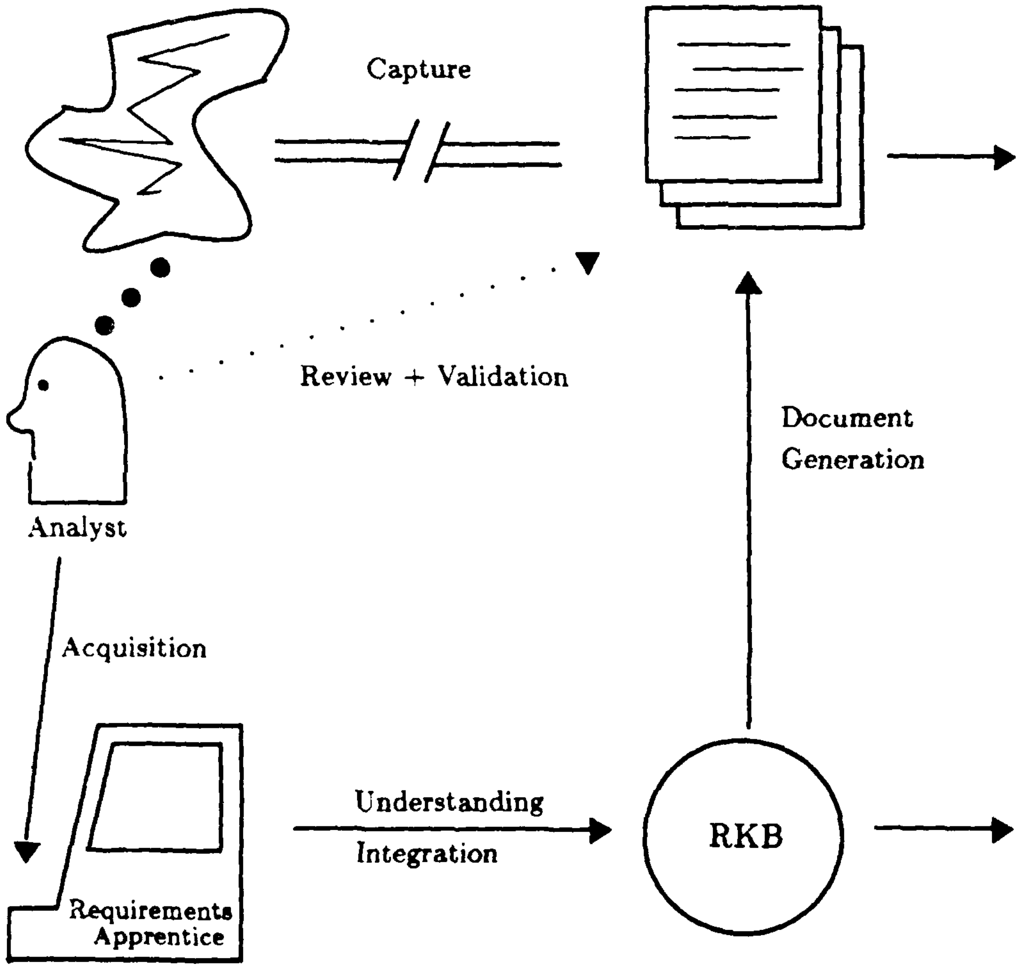

Figure 2-2: The Problem of Requirements Acquisition

the model. The analyst provides these statements one at a time, trying to cess them in a sensible coherent order. However, the analyst is not infallible. He even may include distorted statements that do not fit the model.

As the analyst makes statements, the RA's goal is to narrow in on a model for those statements. The RA's task is to prune the set of models consistent with the input down to a single model. This is the process of requirements acquisition. Informality arises in trying to recover information from the presentation of partial input. Note, however, that the input language to the RA is formal, a restricted form of first order predicate calculus. It is not the source of informality. The informality is of a semantic nature due to fragmentary and ambiguous input.

It may be the case that the input to the RA is sufficiently distorted so that the RA converges upon a model different from that in the client's head. This risk is what the process of validation is designed to discover. Unfortunately, there is reason to believe that validation is, in general, an unsolvable problem [13].

An important aspect of this acquisition process is that it strongly supports the notion of the captured requirement as an evolving entity. Most, if not all of the time, the RA cannot prune the set of possible models it is considering to a single one. It must constantly maintain state information that represents the space of possible models determined by the current input. From the RA's point of view it does not matter whether future input deviates from past input due simply to the client's recognition of a previously incorrect statement or due to a change in the model to be captured from the client's head. In either case, the RA must be prepared to alter its conception of what the current model might be.

Finally, by viewing the RA in this light, it can be seen as two different kinds of acquisition devices. First, at the most basic level, the RA is a logical specification notepad. It processes the client's statements in FOL and makes the appropriate logical deductions resulting in either a contradiction or the addition of new axioms that may narrow the current set of possible models.

Second, the RA performs deeper processing by incorporating a source of expectations and definitions in the form of both a domain model and general requirements clichés. The domain model is a large source of additional data to the basic logical processing. It also provides a space of alternatives for the RA to consider in pruning the model space.

2.4 The Structure of Requirements

A model (in the generic sense) of the structure of a requirement is necessary to assist and guide the RA as it acquires a requirement. This same model is also used to guide the production of a requirements document. The model outlined below sets up some basic expectations on the contents of the requirements input description. It also begins to define a methodology for using the RA. The model defines three types of information that need to be described in a requirement: *environment*, *needs*, and *system*. The environment and needs define the problem whose solution is described by the system.

The environment defines the set of external behaviors the system will have to deal with and the objects and agents generating these behaviors. The environment description captures the immutable reality in which the system will be implemented and operating. The environment exists (e.g., "the hospital in which there are patients, who are sick, and doctors, who make them well"). Initially, the analyst may make changes and refinements to the environment description as his understanding improves. It is thought of as immutable, however, in the sense that any change to this description entails a significant commitment, i.e., to a change in the real world.

The environment is described as a complex artifact with agents that interact with objects via actions to produce behaviors that characterize the environment. This environment is the generator of behaviors that the system must respond to. Any assumptions or simplifications to this description should be introduced into the needs description.

The idea of modeling the environment a system is to operate in is present in work on specification [9, 14]. The motivation is that in order to define the system, in particular its interface, a universe of possibilities must be defined. Given such a definition, system definition can be thought of as defining a hole inside the universe defined by the environment description (see figure 2-3). Without a containing boundary, system definition is unrestricted.

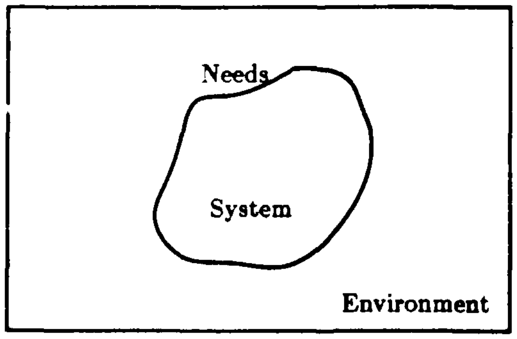

Figure 2-3: Defining a System

Needs describe a viewpoint which defines what aspects of the environment comprise the problem. Needs are perceived. The same environment description could have many different needs descriptions associated with it, each defining a different requirement. Needs are negotiable, particularly when considering costs and risks of satisfying a proposed need. The needs description will evolve as the problem becomes more clearly understood.

Needs can be expressed in terms of product functions, policies, or problems. Product functions abstractly describe an operation that the system will have to perform, e.g., "provide data about current hospital patients." Policies declaratively state properties which the system must maintain, e.g., "patient records should be private." Problems state unstructured system needs.

The system is a description of the solution that will be built to solve the problem described in the needs section. This solution is described primarily in terms of *functional requirements*, specific functions the system must be able to perform. A combination of functional requirements (or possibly a single one) account for a product function or enforce a policy. For example, performing the product function that requires "providing data about current hospital patients" may use the functional requirements which: "store patient data," "retrieve patient data," and "prune data for privacy."

Using these three kinds of descriptions, a requirement acquired by the RA can be seen as: defining the context within which a system will have to be built, stating the problem to be solved, and sketching an initial high level solution to the problem.

A purist might think that a requirement should only describe what the problem is without any reference to how it is to be done. If so, the existence of the system description in RA requirements could be questioned. The solution description is included for a number of reasons. Any practical requirement needs to include some guidance as to the kind of solution desired. The client often will not wish to give the vendors carte blanche. It is also the case that people do not naturally separate the description of 'what' from 'how' and will present problem descriptions in terms of solution technology. Perhaps people should be trained to be more disciplined about their levels of description, but there is an argument that points to the need for some level of 'how' description in requirements. Consideration of solution level decisions often reveals issues that are of general concern but would not have come up at a more abstract level of description. Finally, by incorporating an initial solution sketch, the RA can provide input to high-level design and specification tools, bringing us closer to automated assistance for the whole software development process.

Validation

The RA provides little explicit support for validation. It does, however, set up a framework in which validation can proceed. Validation consists of two tasks. First, the set of needs must be correct, i.e., no missing needs and no extraneous needs. The reuse of requirements clichés will add certain needs as appropriate. Some extraneous needs may be discovered through conflict detection. Second, each need must be accounted for by some aspect of the system which will satisfy that need. For example, each product function has to be satisfied by one or more functional requirements that can be combined to achieve the desired behavior. This satisfaction relation can be represented in the requirements knowledge base. The underlying dependency maintenance can be used to monitor the satisfaction of the entire set of needs. This framework sets up a recording capacity for the decisions required to validate the requirement. The actual reasoning that decides a particular system component(s) satisfies a particular need is unsupported except for clichéd solutions captured in the relevant pieces of the domain model.

Accountability

Another reason for dividing information into environment, system, and needs is to support a separation of concerns which can assist in creating higher quality and more reliable software. First, by explicitly separating concerns the clients are forced to distinguish features of their current situation, from desired features of the new system, from the actual needs of the software project. It is important to maintain a record of the ideal requirement before diluting it with practical compromises [15]. In this way, when new technology comes along, it is possible to improve the system. For example, suppose a system is being created that needs to monitor an analog variable. Ideally, this variable should be continuously monitored, but for technological reasons it can only be monitored at 25 MHz. The need should be stated as continuous monitoring with the comprise to 25 MHz explicitly recorded. This enables the sampling rate to be updated as conditions change.

Separating concerns also provides greater accountability by providing a division of blame for system errors. An error may be due to an incomplete understanding of the physical world. A famous example of this is when the ABM warning system detected a large number of missiles heading towards the United States [13]. The problem was a false radar image from the rising moon. Apparently, the moon was not modeled as a data source in the environment. In this case, the problem can be traced to the environment description, fixed, and made available for future software developments. This achieves two important goals. First, the environment models get asymptotically better. Second, all systems derived from a particular environment model can benefit when a bug is found in the course of operation of any one of the systems. The correction to the environment model may trigger rework in the other related systems.

An error may be due to mistaken needs. A flawed validation process accounts for such errors. Perhaps in such a case, blame rests with the client. Validation can never provide a guarantee, however, and needs are actually a target that moves with time. Errors in this part of a requirement may be unavoidable.

Finally, the error may be in the system model, in which case blame may reside with the software vendor. While this division may be a naive first step, it will certainly be of concern as long as requirements fill a contract role in software development.

2.5 A Requirements Ontology

In addition to the basic environment, needs, and system division, the RA makes use of an ontology of standard requirements objects. This ontology provides a basic set of objects to be used in structuring the dialog with the RA and in its internal reasoning. Each object has a definition associated with it. The combined effect of having different object types and associated definitions is to permit some basic consistency checking of the requirement. In essence, it provides a form of type checking that helps detect blatant category errors.

Operations are processes. They have inputs and outputs. Operations also have pre- and post-conditions which are true of every invocation of the operation. Normal-pre and normal-post conditions capture the behavior of the operation during non-exceptional executions.

An *Action* is a state transition operator. It is an atomic transition. In the absence of any other information, the behavior of an environment (or a system) could be simulated by a non-deterministic sequence of invocations of actions that the agent is capable of executing. Actions are operations and thus should be simulable.

It should be noted, that the RA is very weak on analyzing the operational behavior of an artifact. For example, it cannot simulate test scenarios. The semantics of the operational notions is presented to motivate their inclusion in the ontology and to point to places where operational analysis can be integrated into the RA. The level of analysis that is performed on such objects is at the level of answering: What operations exist? and Who executes them?

A *Behavior* is composed of a number of actions executed in some programmed sequence to achieve a certain purpose. A behavior represents a typical pattern of state change in an environment (or system).

An *Agent* is an entity that can execute behaviors. Agents are the actors in any execution sequence. Agents are associated with either an environment or system.

An *Invocation-Protocol* describes which agents typically execute which behaviors.

A *Product-Function* is a kind of need. It describes a functional ability the system must have.

A *Policy* is a kind of need. It constrains the behavior of the system.

A *Problem* is a general kind of need. It defines a need which the system must provide a solution to.

A *Functional-Requirement* is an operation that the system performs, presumably to satisfy some need, e.g., a product-function. A functional-requirement is a kind of operation and thus has an operational definition.

A *Solution* is a manner of solving a problem.

This ontology provides a set of conceptual primitives to be used in elaborating environment, system and needs descriptions. An environment (or system) is an entity (an agent) with associated agents that perform behaviors. Each behavior performed by the system has a purpose in fulfilling a need.

Similar Requirements Models

The RA is one of the first systems to integrate a model of requirements descriptions with active knowledge-based reasoning and support. The model and ontology proposed above are not a radical departure from current models of the requirements process. Influences from a number of sources (e.g., [16, 17]) are present:

ANSI/IEEE Standard 830-1984 [18] proposes a requirements outline that includes the notion of separating needs and system descriptions. The standard does not emphasize the separate description of the environment, though such a division is consistent with it. Of course, this standard also includes many sections that are necessary for management and implementation of a software system that are not included here because of the RA's focus on acquisition of informational requirements.

Potts's and Finkelstein's work on the FOREST project [19, 20] provides a closed system viewpoint on requirements description in addition to illustrating the different sort of components (actions, agents, and dataflow) in such a description. This closed system viewpoint is shared by GIST [21] which also incorporates the notion of agents being generators of behaviors.

Matsumura's et al. work [22] echoes the importance of viewpoints, the structuring that needs (product functions) provide for system functional requirements, and the process of enumerating and refining terms to define a domain of discourse.

Greenspan's work on RML [23, 24] defines an extensive ontology to be used in representing a requirements description, particularly the environment (world model) description. RML is perhaps the first application of AI technology to requirements acquisition. It concentrates on designing and formalizing a language in which to represent requirements and on describing a methodology to guide the use of this language. Again, the RA is one of the first systems to attempt automated assistance for the acquisition processes.

2.6 The Requirements Document

A session with the RA provides the analyst with feedback as the description is presented. After the dialog is complete (or suspended) the entire session is captured in the RKB. While the RKB is useful for transferring information to other tools, it is not useful for communication with people (e.g., for review of the requirement or for evaluation of this research). The RA can generate a requirements document which captures the current state of the acquisition.

This requirements document is a window into the RKB. Its production is guided by the RA's requirements model. There is a separate section in the document for each of the environment, needs, and system. An introductory section provides a brief, high-level description of the requirement.

The requirements document is produced by describing the cliché instance that is the requirement. (There is a cliché in the reference library called 'Requirement'.) This serves as a root that permits access to most of the information about the requirement. Another section in the requirements document captures the loose ends that represent the state of the acquisition. This section includes information about exceptional facts, pending agenda items, incompleteness (missing values), and currently outstanding contradictions.

The structure of the requirements document is guided at the top-level by the RA's ontology of requirements acquisition. At lower levels its organization is guided by the data specific to the current requirement. Sections are automatically generated for important aspects of the current requirement. The data to fill these sections is extracted from the RKB and grouped together. Each of the three main sections also has a dictionary at the end which assists in the understanding of the document.

2.7 Domain Modeling

The RA's operation is greatly enhanced by reference to a domain model of the requirement being acquired. As pointed out previously, the RA can operate in a minimal fashion as a logical specification notepad without a domain model. The notepad verifies the internal consistency of data presented. Could the RA operate starting with an empty domain model and still achieve a level of understanding which includes the resolution of informalities? It is unlikely. Consider the following requirements sketch from [12]:

> "Given a frompie consisting of frangs separated by Plunts or by New-Swintzel tranlings, convert it to a swintzel-by-swintzel form in accordance with the following rules:
> 1. swintzel breaks must be made only where the given fromple has Plunt

> or New-Swintzel,
> 2. each swintzel is filled as far as possible, as long as
> 3. no swintzel will contain more than MaxPos tranlings."

All the domain dependent nouns have been replaced by nonsense equivalents. The verbs have been left in, though they also should have been replaced since they carry as much information in their spelling as nouns. There is some information in the text of the description, but there is no explicit hook to background information which permits debugging the description or exploring related issues.

When the following pairs of substitutions are made the text is decoded: text/frompIe, line/swintzel, word/frang, blank/plunt, character/tranling.

> "Given a text consisting of words separated by Blanks or by New-Line characters, convert it to a line-by-line form in accordance with the following rules:
> 1. line breaks must be made only where the given text has Blank or New-Line,
> 2. each line is filled as far as possible, as long as
> 3. no line will contain more than MaxPos characters."

The RA relies on the existence of a domain model to achieve performance that may surprise the analyst with relevant data and criticism that were unforeseen. The requirement is built by using pieces of the domain model as building blocks. Each component of the domain model provides data about a particular concept. Associated concepts can provide data which can anticipate problems and changes in an evolving requirement.

The next section contains an extended, annotated example of the RA at work on the "University Library Database Problem" [7]. The analyst uses certain terminology in the example that comes from an understanding of the following domain model. Note, that some of the terminology used also comes from an understanding of the model of requirements and the requirements ontology.

Figure 2-4 shows a schematic outline of part of the contents of the RA's cliché library. The library contains clichés relevant to the three principal aspects of a requirement discussed above.

The domain model of environments consists of simple categorical information about things like physical objects and containers. More detailed information is present about a *repository*. A repository is an entity in the physical world. The basic function of a repository is to ensure that items that enter the repository are available for later removal. The repository cliché has a number of roles including: the *collection* of items stored in it, the *patrons* that utilize the repository, and the *staff* that manages the

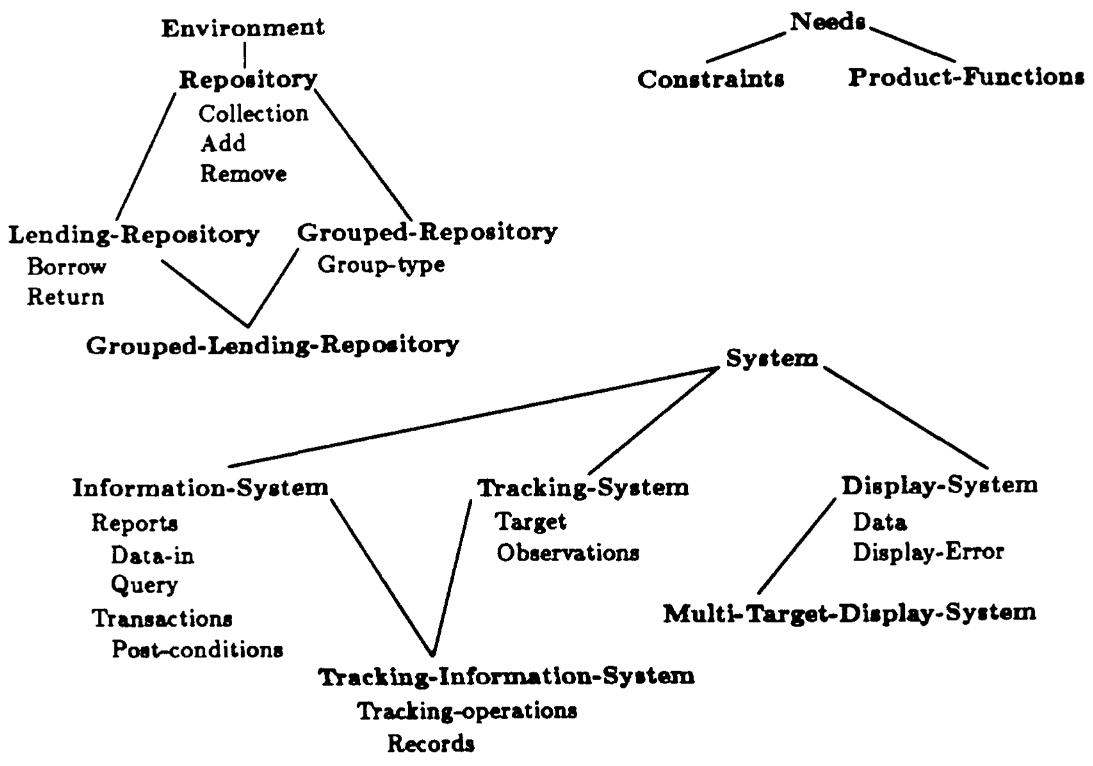

Figure 2-4: Part of the Requirements Cliché Library

repository. The two key operations on a repository are *adding* and *removing* items. There are a variety of physical constraints that apply to repositories. For example, since each item has a physical existence, it can only be in one place at a time and therefore must either be in the repository or not.

There are several kinds of repositories. Simple repositories merely take in items and then give them out. *Lending repositories* support the borrowing of items, which are expected to be returned. *Grouped repositories* contain items aggregated into groups of similar items. Example repositories include: storage warehouses (simple repositories for unrelated items), grocery stores (simple repositories for items grouped in classes), and rental car agencies (lending repositories for items grouped in classes). Finally, a *grouped lending repository* is the conceptual combination of grouped repository and lending repository.

In the area of systems, the domain model includes knowledge about *information systems* which are transaction processing systems that produce reports and maintain certain data invariants. The intent of the information system cliché is to capture the commonality between programs such as personnel systems, bibliographic data bases, and inventory control systems. The central roles of an information system are a set of *reports* that display parts of the data being stored and a set of updating *transactions* that create/modify/delete the data. Each report and transaction is in turn a cliché with its own characteristic roles. Additional roles of an information system include: the *staff* that performs the transactions and the *users* that obtain the reports.

A *tracking system* keeps track of the state of some *target*. It does this by making *observations*. The observations can either sense the state of the target directly (e.g., the way a radar keeps track of the position of airplanes) or observe the operations that change the state (e.g., the way a set of turnstiles keeps track of the number of people in a building).

The *tracking information system* cliché combines the features of an information system and a tracking system. The main content of this combined cliché is a set of constraints that relate roles of the information system part to the tracking system part. For example, the tracked data role becomes the data in the information system. Further, the observations in the tracking system correspond to transactions in the information system.

A number of other miscellaneous generic concepts are represented. The notion of inverse is understood to a fair degree, including a notion of compatibility of pre- and post-conditions. Models for the state that objects can be in with respect to a container are represented and include a five, three, and two state model.

The outline of concepts modeled in the domain cliché libraries gives an idea of the level and breadth of the knowledge the RA has before tackling the library database scenario. The following scenario will describe a system by building on top of the concepts enumerated above.

2.8 An Example: The Library Database System

The library database problem (see figure 2-5 on the following page) [7] is one of the (defacto) benchmark problems for specification and requirements tools. It was used by Kemmerer in work on testing specifications [10]. The problem was then adopted at the 2nd International Workshop on Software Specification and Design (IWSSD)[1] to evaluate tools submitted in a tools session. The 4th IWSSD included it as one of four problems [25] that participants were to use as examples in their papers. An article by Wing [26], based on work presented at IWSSD4, reports on 12 papers that analyzed the library problem.

The statement of the library problem is deceptive. It may appear unrealistic. It is stated in one page of text. What realistic problem can be described so succinctly? Nevertheless, its description leaves ample room for interpretation and many issues are unresolved as demonstrated in the following scenario. The scope of the problem is limited though it does describe the basic capabilities of a library database system. No one has yet created a system to "solve" the library problem in any comprehensive fashion. This indicates that it remains a good exercise for software creation tools.

A potentially more serious problem with the library example is that it describes a tracking system, a system that echoes changes in the environment. Missing from the problem description is new functionality and computation that does more than mirror the environment, e.g., a requirement to keep the inventory of books on hand adequate. This can be seen by noticing that most of the description can be viewed as a description of the environment with a simple requirement to track the environment. Knowledge acquisition figures prominently in obtaining an understanding of the environment so that it can be adequately modeled. The most strongly demonstrated function of the RA is in its knowledge acquisition role. The RA's support for requirements analysis (e.g., do we need a function to keep inventory adequate, what data is required to compute such a function, how will the function be realized in the system) is less strongly demonstrated.

All four problems given in the IWSSD4 problem set [25] share this common problem, i.e., a reliance on one kind of analysis and thus an apparent loss in generality. The heating system problem is basically a description of a finite state machine. The formatter problem is a constraint satisfaction problem. The lift problem is a finite state machine and constraint satisfaction problem. This points to the fact that general analysis techniques need to be supplemented with tools that have detailed knowledge about particular problem formalisms and analyses. Nevertheless, acquiring the data to populate the formalism remains an important problem to be addressed by general

[1]which, at the time, was called Workshop on Models and Languages for Software Specification and Design.

Consider a university library database. There are two types of users: normal borrowers and users with library staff status. The database transactions are:
(1) Check out a copy of a book.
(2) Return a copy of a book.
(3) Add a copy of a book to the library.
(4) Remove a copy of a book from the library.
(5) Remove all copies of a book from the library.
(6) Get a list of titles of books in the library by a particular author.
(7) Find out what books are currently checked out by a particular borrower.
(8) Find out what borrower last checked out a particular copy of a book.

These transactions have the following restrictions:

- R1 - A copy of a book may be added or removed from the library only by someone with library staff status.
- R2 - Library staff status is also required to find out which borrower last checked out a copy of a book.
- R3 - A normal borrower may find out only what books he or she has checked out. However a user with library staff status may find out what books are checked out by any borrower.

The requirements that the database must satisfy at all times are:

- G1 - All copies in the library must be checked out or available for checkout.
- G2 - No copy of a book may be both checked out and available for checkout.
- G3 - A borrower may not have more than a given number of books checked out at any one time.
- G4 - A borrower may not have more than one copy of the same book checked out at one time.

Figure 2-5: The Library Example (c) 1985 IEEE

purpose knowledge acquisition tools such as the RA.

The scenario described below demonstrates the RA as it acquires a requirement for the library database problem. It is an annotated trace of the RA as it ran on May 2, 1990. When looking at the trace, it is important to focus on its content, not its form. The content illustrates the fundamental capabilities of the RA. However, the form is at best a weak shadow of what the dialog portion of the interface should be like. The particular commands shown were chosen based on the library data base problem. While adhering to the concept outlined in the problem statement, the input makes certain aspects of the problem more explicit and expands on the problem definition.

Paraphrasing

In the following dialog, input to the RA is in `normal size typewriter font` and preceded by a command number such as: `17>`. Output generated by the RA is in `code size typewriter font` and preceded by an alphabetic number such as `17a:`. Commands to the RA consist of a list of words. The first word is the command name. More details about the command language are discussed shortly. Each command is in a separate section with a title that is descriptive of the main feature demonstrated by that command.

```
1> (Find-Requirement Library-System)
```

```
1a: Beginning-A-New-Requirement-Called-The Library-System.
1b: Library-System Is-An-Instance-Of Requirement.
```

The analyst opens a session by requesting access to a named requirement, e.g., Library-System. The RA searches its known knowledge bases and either resumes an analysis from its saved state or, if no knowledge base is found, begins a new requirement. In actuality, the current version of the RA cannot save an analysis due to lack of this capability in the underlying reasoning system. Implementing such a capability is not difficult. In fact, a general Lisp checkpointing system is being developed, by a member of the Programmer's Apprentice project, that will take as input a description of the program state and generate a dump of the current state of a program. A more pragmatic problem is the fact that the dump of the reasoning system is likely to be multiple mega-bytes in size. If the RA were used over an extended period of analysis on a project, the size of the knowledge base would be huge. Research will need to be done that extends the work in handling very large databases to very large knowledge bases.

The RA acknowledges every input with a paraphrase of that input and any interesting deductions made due to the added knowledge. The paraphrase allows the analyst to judge whether or not the RA has interpreted the command correctly. The other deductions alert the analyst to implications of his statements. In this case, the paraphrase 1a imparts information implicit in the the input that the requirement is new. The deduction 1b seems redundant with the paraphrase. It is present since the requirement itself, i.e., Library-System, is a full fledged object in the knowledge base about which things can be discovered, i.e., it is of type requirement.

The RA decides what to say in response to a command by monitoring changes in the requirements knowledge base. Finding things to say is not so much the problem as deciding what *not* to say. The underlying reasoning system typically deduces an avalanche of facts, only a few of which are worth reporting. The RA collects a list of all the new and changed facts matching particular syntactic forms. It then filters these facts further to avoid redundancy. For example, if an object is deduced to belong to several different types, all but the most specific type assertions are pruned away. The RA then displays the remaining facts in an order based on the objects they refer to. (For a further discussion of the problems involved in choosing an appropriate level of explanatory detail see [27].)

The Command Language

```
2> (Define University-Library-Database :System :Synonym Uldb)

2a: Uldb Is-A-Member-Of Library-System.System.
2b: Uldb Is-An-Instance-Of System.
```

With the requirements model in mind, the analyst identifies the system being described by giving it a name. The RA's command language is based on an entity-relationship model. The `Define` command creates a new entity. The first argument is the name of the entity being defined. This name is often followed by one or more keywords declaring the type of the entity. The remainder of the arguments to the define command are pairs of keywords and values that identify particular features of the definition. In this case, a short synonym (ULDB) is defined that can be used on input and output when referring to the university library database. Output 2a reflects the RA's definition of a requirement. The requirement has a system role and the ULDB fills this role. (The syntax "X.Y" is used to refer to the Y role of X.) Output 2b is a paraphrase of the input.

New Terms are Quoted

```
3> (Define University-Library :Environment :Library :Synonym Ul)

3a: Ul Is-A-Member-Of Library-System.Environment.
3b: Ul Is-An-Instance-Of "Library".
3c: Ul Is-An-Instance-Of Environment.
```

Still mindful of the requirements model, the analyst identifies the environment of the requirement as a library. In the output produced by the RA, the word library appears in quotes (3b). This signifies that although the RA has recorded that the

UL is a library, it does not know what a library is. Words in the paraphrase are surrounded by quotes unless they are defined by requirements clichés or have been explicitly defined by the analyst in define commands. This is done to indicate that this term must eventually be defined (to help in catching typographic errors).

In this scenario, it is assumed that the RA has extensive knowledge of information systems, tracking systems, and repositories but no knowledge about libraries or library information systems per se. To operate in the intended manner, the RA must have a considerable amount of background knowledge relevant to the requirement at hand. However, it is unrealistic to assume that this knowledge will ever be complete.

Output 3a parallels 2a as the UL is identified as the environment of the requirement. In 3b and 3c both types are stated since neither subsumes the other (which would permit the pruning of the more general type).

Simple Disambiguation

```
4> (Need (!Logic (Tracks Uldb Ul)))

4a: (!Logic (Tracks University-Library-Database University-Library))
     Is-A-Member-Of Library-System.Needs.
4b: Uldb Is-An-Instance-Of Tracking-System.
4c: Uldb.Target Has-Value Ul.
```

Now the basic need between the environment and the system is stated. When the first word of a command (e.g., Need) is unknown, the command is interpreted as a logical assertion. This extends the entity-relationship data model with general logical assertion. The form (!Logic ...) is used to quote the body of a logical formula for printing purposes, semantically it is the same as if the !Logic had been omitted.

An interesting aspect of this command is the use of the word tracks. The way the RA attaches a meaning to this word in this context is a simple example of what goes on in the RA in general when it tries to interpret the analyst's statements. The relation tracks has several specializations in the cliché library, e.g., tracks-tracking-system and tracks-audit-system, and thus the use of the word tracks is ambiguous. The specialization of tracks that is consistent with what is known about ULDB and UL is associated with a specialization of a particular kind of system called a tracking-system. In the current assertion, the word tracks is refined to 'tracks-tracking-system', partially resolving the ambiguity. As a result of propagating the refined definition, the RA deduces that the ULDB must be a tracking system (4b). This is reported to the analyst, who could choose to override the assumption.

At this point, the RA has not succeeded in totally disambiguating what tracks means. It has located an intermediate specialization of the term, which reflects its intermediate level of understanding. The cliché library is designed to be a multiple hierarchy with many intermediate terms that can be used in this way.

Output 4c shows a deduction based on the definition of tracks-tracking-system and terminology associated with tracking systems. In this case, the deduction is simply an alternative indexing of relationships, i.e., the target of the ULDB is the UL.

An important goal of the RA is to leave the analyst in control of the interaction. The RA is designed to be non-invasive in that it primarily just processes what the analyst says without asking questions, even if it is not able to understand these statements completely. However, the RA maintains a list of issues that need to be resolved. This information should be displayed to the analyst in an unobtrusive way as a list of pending issues (see figure 2-6).

After this command, four issues are pending. Two of them are stored on the agenda and the other two are stored as part of the processing state of the RA. The word library needs to be defined (item 1). The word tracks needs to be further disambiguated. The fact that ULDB tracks UL means that there must be some state inside UL that is being tracked (item 6). The composition of the UL state and the parts of it tracked by the ULDB need to be defined.

```
1. New-Term: Id= (New-Term . Library) (P=2).
6. Make-Defined-Term: Id= (Defined-Exp Item-States !Args University-Library)
                      (P=1).
```

Figure 2-6: Pending Agenda Items

This list of pending issues indicates one of the ways that the RA can assist in ensuring that a requirement will be complete. Terms mentioned by the analyst access clichés, which contain various places where additional information has to be filled in and which also instantiate their respective definitions (which in turn may access other clichés). Filling in this additional information makes the requirement more complete but may also access additional clichés. This process continues until a point of closure is reached where filling in the remaining information does not lead to the inclusion of additional concepts with loose ends.

Defining a New Term

```
5> (Define Library :Ako Repository :Defaults (:Collection-Type Book))

5a: Ul Is-An-Instance-Of Repository.
5b: Ul.Collection-Type Has-Value "Book".
5c: Patrons Of Ul Is-An-Instance-Of Interacting-Agent.

5d: Library Is-A Environment, Container, Physical-Object, Agent, Repository.
5e: Library  Has-No-Local-Slots.
    And-Inherited-Slots Product-Functions, $Behaviors, $Agents, $Actions,
    State-Of, Item-States, Patrons, Staff, Storage-Location, Collection,
    Collection-Type.
```

The analyst defines a library to be a kind of (:AKO, i.e., a specialization of) a repository that by default contains books. The RA knows what the general concept of a repository is, but does not know what a book is. The definition of repository is propagated to the UL. This definition includes information about which agents interact with the repository. In this case, they are simply called patrons (5c). This command answers two of the pending issues mentioned above and adds a new issue regarding the new term book (see below). As well as defining the word library, it describes what the state inside the UL is, i.e., the set of books contained in it (5b).

```
7. New-Term: Id= (New-Term . Book) (P=2).
```

Figure 2-7: Later Pending Agenda Items

This command defines a class of objects called libraries that inherits properties from the repository cliché. Is library itself a requirements cliché? The concept of a library could be described as a requirements cliché. There certainly is a rich body of information about libraries that could be captured. The definition provided interactively, however, contains only a small fraction of the data necessary to capture the clichéd notion of a library. This brings up an interesting question again. If the RA started with an empty domain model, could it still acquire the library system requirement? To do so would require a much greater amount of input from the analyst. The command language is not currently robust enough to permit stating the full range of things that can be captured in a requirements cliché definition, though it could easily be extended to permit such statements. In principle, therefore, it should be possible to acquire any requirement from scratch, albeit at a much greater effort. Library happens to be represented in much the same way as the repository cliché but this overlap of representation should not be construed to imply that library is a cliché. For a concept to be considered a cliché, its definition must be well thought out and integrated in the cliché libraries. A cliché represents a recurring pattern of behavior or reasoning. The key aspect is the repeated pattern. Naming the cliché does not capture its pattern of activity. The RA begins its analysis with a comprehensive domain model provided by the cliché libraries.

Another New Term

```
6> (Define Book :Ako Physical-Object :Member-Roles (Title Author Isbn))
```

```
6a: Book Is-A Physical-Object.
6b: Book Has-Local-Slots "Isbn", "Author", "Title".
    And-Inherited-Slots State-Of.
```

Book is defined to be a class of physical objects with title, author, and ISBN roles. Member-roles describes which roles members of a class will have. Notice that the quotes around book have disappeared with its definition (6a).

Defining a New Role

```
7> (Define Book.Isbn :Ako Integer :Cardinality Single :Unique-Id T)
```

```
7a: Book.Unique-Id Has-Value Isbn.
```

In addition to entities, class roles can be defined using a dotted first argument to the define command. The ISBN of a Book is a role that can only contain a single integer value, i.e., its cardinality is one. This value uniquely identifies a book (:Unique-Id).

Changing Contexts

```
8> (Go-To-Context Functional-Requirements)
```

```
8a: Expecting-Definition-Of Functional-Requirements Of Uldb.
```

One technique of informal communication is to leave out pieces of the description that are obvious from context. The Go-To-Context command provides a way to say "now lets talk about this." The RA maintains a stack of explicitly entered contexts. Another plausible alternative would be for each command to implicitly define one or more contexts and to augment the context stack with these implicitly referenced contexts.

The context a statement is made in is used to fill in missing details. Part of the fragmentary nature of input statements is that certain role values, obvious from the context, are left out by the speaker. The context mechanism helps in deducing these missing values. Ideally, informality resolution should occur relative to the current context, i.e., the context could be used to propose resolutions and to order alternative resolutions.

In the Listener, contexts are simply roles of a cliché instance. While in a particular context (8a), any objects defined that are of the appropriate type to be members of that role are made members (as will be seen in the following commands). Once an object becomes a member of the context role, the containing instance (e.g., ULDB in the context ULDB.functional-requirements) becomes a source for the propagation of data to the object being defined.

A Highly Underspecified Definition

```
9> (Define Check-Out :Roles (:Records Remove))

9a: Uldb Is-An-Instance-Of Tracking-Information-System.
9b: Uldb.Users Is-An-Instance-Of Interacting-Agent.
9c: Uldb.Manner-Of-Observation Is-An-Instance-Of Indirect-Observation.
9d: Uldb.Data-Observed Is-An-Instance-Of State-Changes.

9e: Check-Out Is-A-Member-Of Uldb.Functional-Requirements.
9f: Check-Out Is-An-Instance-Of Action-Tracking-Operation.
9g: Check-Out.Objects Has-Value
    (!The-Set-Of-All (?O) Such-That (= (Isbn ?O) $Input)).
9h: Check-Out.Target Has-Value Ul.
9i: Check-Out.Object-Type Has-Value Book.
9j: Check-Out.Normal-Post Has-Value
    (!Text "(Not (Mem-repository ?Item University-library))").
9k: Check-Out.Normal-Pre Has-Value
    (!Text "(Mem-repository ?Item University-library)").
9l: Check-Out.Pre Has-Value
    (!Text "(And (Book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent)))").
9m: Check-Out.Input Has-Value (!Text "(?Operator ?The-agent ?Item)").
9n: Check-Out.Records Has-Value Remove-Repository.
9o: Check-Out.Records-Function Is-An-Instance-Of Remove-Repository-Action.
```

Having defined the basic environment of the library system, the analyst now begins the definition of functional requirements. Interpreting this command presents a significant challenge to the RA because no type is declared and, while the words records and remove are both in the requirements library, neither one has a unique definition. This problem is resolved by attempting to locate something that can 'record' 'remove' and is meaningful in the context of a tracking system.

Check-Out is determined to be of type action-tracking-operation (9f) since tracking operations 'record' things and action tracking operations record actions (remove is an action). Tracking operations are a kind of transaction and transactions are defined in information systems. Since Check-Out is a functional requirement of the ULDB (9e), the ULDB is determined to be an information system. Since the ULDB is already known to be a tracking system its type is further refined to be a tracking information system (9a).

The ULDB is a tracking system whose job is to track the state of its target (the UL). It tracks the state by tracking state change operations (like remove). This allows the RA to conclude that the kind of data observed is state changes (9d). The manner of observing the state of the system is indirect observation (9c), because the system is not actually monitoring the state, rather it is computing state changes based on its knowledge of an initial state and the effects of state transition operators.

The information in the action tracking operation cliché, along with the current RKB permits the RA to deduce a significant amount of information about Check-Out. For example, since the ULDB tracks the UL, and since the UL is a library that stores books, Check-Out must keep track of objects of type book (9i). A default constraint in the action tracking operation cliché declares that, in the absence of anything else being said, an input of Check-Out should be a unique identifier for the kind of object being tracked and that the objects being tracked are the objects associated with this unique identifier (9g).

The word remove can be understood because it corresponds to one of the fundamental operations that can alter the state of a repository (9n). Understanding remove as remove-repository permits the propagation of pre and post conditions to the Check-Out transaction (9j,9k,9l). The input specification (9m) is derived from remove repository and augmented with an **?Operator** argument to reflect the fact that this state transition is being recorded by an operator of the tracking system.

The simple definition of Check-Out has turned out to contain a great deal of implicit information. The RA discovers this information through its attempts to understand the input statement. This understanding involves disambiguation, refinement, and plausibility checking all acting together. The results are presented to the analyst.

A Brief Description

```
10> (Inverse Check-In Check-Out)

10a: "Check-In" Is-A-Member-Of Uldb.Functional-Requirements.
10b: "Check-In" Is-An-Instance-Of Action-Tracking-Operation.

10c: "Check-In".Normal-Post Has-Value
      (!Text "(Mem-repository ?Item University-library)").
10d: "Check-In".Normal-Pre Has-Value
      (!Text "(Not (Mem-repository ?Item University-library))").
10e: "Check-In".Pre Has-Value
      (!Text "(And (Book ?Item)
                   (Or (Staff ?The-agent) (Patrons ?The-agent)))").
10f: "Check-In".Input Has-Value (!Text "(?Operator ?The-agent ?Item)").
10g: "Check-In".Objects Has-Value
      (!The-Set-Of-All (?O) Such-That (= (Isbn ?O) $Input)).
10h: "Check-In".Object-Type Has-Value Book.
10i: "Check-In".Target Has-Value Ul.
10j: "Check-In".Records Has-Value Add-Repository.
```

The analyst gives a quick definition of the Check-In operation by stating that it is the inverse of the Check-Out transaction (i.e., it tracks the inverse state change). From this statement, the RA is able to derive quite complete information about Check-In. Starting with the command 9, a cluster of information in the RKB has reached a critical mass where the RA can draw much more extensive conclusions from each new related statement.

In particular, because add and remove are inverses, Check-In records add repository actions (10j). This provides the same amount of information about Check-In as command 9 provides about Check-Out, therefore, a parallel set of deductions is made.

An Accidental Alignment

```
11> (Define Acquisition :Roles (:Records Add))

11a: Acquisition Is-A-Member-Of Uldb.Functional-Requirements.
11b: Acquisition Is-An-Instance-Of Action-Tracking-Operation.

11c: Acquisition.Target Has-Value Ul.
11d: Acquisition.Objects Has-Value
     (!The-Set-Of-All (?O) Such-That (= (Isbn ?O) $Input)).
11e: Acquisition.Object-Type Has-Value Book.
11f: Acquisition.Normal-Post Has-Value
     (!Text "(Mem-repository ?Item University-library)").
11g: Acquisition.Normal-Pre Has-Value
     (!Text "(Not (Mem-repository ?Item University-library))").
11h: Acquisition.Pre Has-Value
     (!Text "(And (Book ?Item)
                  (Or (Staff ?The-agent) (Patrons ?The-agent)))").
11i: Acquisition.Input Has-Value (!Text "(?Operator ?The-agent ?Item)").
11j: Acquisition.Records Has-Value Add-Repository.
11k: Acquisition.Records-Function Is-An-Instance-Of Add-Repository-Action.

Conflict #1
Colliding-Definitions-Of Acquisition And ''Check-In''
Both-Are-Realized-As-Equivalent-Instances-Of Action-Tracking-Operation
```

The analyst defines an Acquisition transaction that tracks the addition of a new book to the library collection, i.e., not simply a return. The RA is again able to determine significant amounts of information about this transaction and deduces that there is a problem with the requirement. From what has been said so far, the Acquisition and Check-In transactions appear identical. They both record the addition (the inverse of removal) of a book to the library. The RA has a built-in expectation that terms should not be synonymous unless they are explicitly defined to be synonyms. This bias was introduced on the theory that an accidental alignment of this kind indicates that the analyst has likely left out some critical information, or is using some term in an incorrect way.

```
Explanation:
1.1 (Equal-Defs Check-In Acquisition) is True by Modus Ponens from ...
Premises:
1. (= Isbn (Unique-Id (!Type Book)))
2. (User= Add (Records Acquisition))
3. (User= Remove (Records Check-Out))
4. (Inverse Check-Out Check-In)
5. (System University-Library-Database)
6. (Environment University-Library)
7. (Need (!Logic (Tracks University-Library-Database University-Library)))
8. (Library University-Library)
9. (%Is-A Library Repository)
...
```

```
Resolve conflict?(Y or N) Yes.
Would you like to retract a premise?(Y or N) No.
Would-You-Like-Suggestions-For-Alternate-Premises
 (0 is no, # or list of #s) (2 3)
In-The-Premise (User= Add (Records Acquisition))
You-Could-Make-A-Substitution-For-The-Word Add
Should I choose Remove-Repository as a :Substitute-Value(Y or N) No.
In-The-Premise (User= Remove (Records Check-Out))
You-Could-Make-A-Substitution-For-The-Word Remove
Should I choose Add-Repository as a :Substitute-Value(Y or N) No.
Conflict #1.
Colliding-Definitions-Of ''Check-In'' And Acquisition.
Both-Are-Realized-As-Equivalent-Instances-Of Action-Tracking-Operation.
```

The RA reports the problem and, by way of explanation, exhibits a proof of the equality of definitions. An unfortunate aspect of this proof is that, like many machine generated proofs, it is quite large and not particularly easy to understand (most of the proof is elided above). An area where additional tools would be helpful is in providing better ways to present such explanations. A tool is being developed for the CAKE reasoning system that will compress syntactically distracting information from explanations and permit browsing the explanation tree [28].

The output illustrates two ways the RA can assist the analyst in resolving a conflict. First, the RA prints a list of premises underlying the conflict. The premises corresponding to statements made by the analyst are printed first, followed by information that comes from the cliché library. Although there can be a large number of these premises, it is often easier to grasp the problem by looking at the premise list than by looking at the proof. This is done with an eye towards the 'commitment' to a particular premise, e.g., premise 5 (the ULDB is a system) is quite unlikely to be incorrect. Note that the conflict cannot be resolved without retracting or changing one of the premises.

It might be the case that one of the premises is totally wrong and should be discarded. However, more typically, all of the premises are at least partially correct. The RA can be helpful in this situation by providing suggestions for how a premise, chosen by the analyst for attention, might be altered. This is done based on a number of heuristics. For example, if an equality premise states that something is equal to a particular value X, it might be the case that what is really desired is some value X′ that is a close relative of X. In the third premise, instead of remove, the analyst might have really meant some other operation on repositories. The only other operation on a simple repository is add, which the RA suggests as an alternative. Unfortunately, changing remove to add is certainly not the correct way to fix the conflict since this would cause Acquisition and Check-Out to have the same definition.

The analyst decides to ignore the conflict for the time being—it goes onto the list of pending issues—and starts defining some reports the ULDB needs to support. This delay in dealing with the conflict illustrates that the RA always allows the analyst to be in control of what is dealt with and when. It is often a good idea to ignore something for a while until clarifying information is made available.

Deferring a Conflict

```
12> (Go-To-Context Reports)

12a: Expecting-Definition-Of Reports Of Uldb.
```

With the conflict in terminology recorded as an agenda item, the analyst goes on to define some reports that the ULDB will have to produce.

Reports

```
13> (Define Books-By-Author :Report :Roles
          (:Data-In (!Decl ((?A %Author)))
           :Data-Out (!Decl ((?Titles (!Set-Of %Title)))
           :Query (!Logic (= ?Titles (!The-Set-Of-All (?T %Title)
               Such-That (!There-Exists (?B Book) Such-That
               (And (Mem ?B Ul) (= ?T ?B.Title) (= ?A ?B.Author)))))))))


13a: Books-By-Author Is-A-Member-Of Uldb.Reports.
13b: Books-By-Author Is-An-Instance-Of Information-System-Report.
13c: Books-By-Author Is-An-Instance-Of Tracking-Report.
13d: Books-By-Author.Accessed-Information Has-Value
     "Author-of-Book, Title-of-Book".
13e: Books-By-Author.Target Has-Value Ul.
13f: Books-By-Author.Query Has-Value
     (!Logic (= ?Titles (!The-Set-Of-All (?T %Title) Such-That
      (!There-Exists (?B Book) Such-That
       (And (Mem ?B University-Library) (= ?T ?B.Title) (= ?A ?B.Author)))))).
13g: Books-By-Author.Data-Out Has-Value (!Decl ((?Titles (!Set-Of %Title)))).
13h: Books-By-Author.Data-In Has-Value (!Decl ((?A %Author))).
```

The key content of the report definition is the logical expression that is used to fill the query role (13f). Symbols in the expression beginning with '?' denote quantified variables. Variables may be given a type by pairing them with a type, i.e., (?<variable> <type>). The query describes the essential information content of the report. It is the heart of a high-level requirement for a report. In this case, the report provides the "titles of books in the UL by a particular author." The data-in field (13h) states what data is provided by a user to parameterize the report (!decl stand for declaration). The data-out field (13g) states what kind of values will be produced by the report. The query field (13f) describes the functional relationship between the input and output data.

Note that a more comprehensive system would need to include detailed information such as report layout and frequency of production. This information is not within the scope of the RA's analysis capabilities and is therefore not represented.

The query is specified using a set specification. This level of formality is somewhat inconsistent with the rest of the input to the RA. The definition might have simply been stated as uninterpreted text. However, in this case, a formal description is used as a way of specifying which roles and relationships are relevant to producing the report. The important information, which the RA can extract from the query, is that there exists the types title and book, that books have titles (`?B.Title`) (13f) and authors, and that Mem is a relationship between books and repositories. The RA extracts this information by recursively analyzing the logical statement. At this stage, the correctness of the definition is of less importance than the statement of relevant data.

The RA uniformly applies its analysis routines to all statements. In the case of quantified statements, all variables are replaced with representative anonymous individuals to yield quantifier-free statements that can be analyzed.

A Generalization

```
14> (Define Books-By-Topic :Report :Roles
           (:Data-In (!Decl ((?Topic %Topic))
            :Data-Out (!Decl ((?Titles (!Set-Of %Title)))
            :Query (!Logic (= ?Titles (!The-Set-Of-All (?T %Title)
               Such-That (!There-Exists (?B Book) Such-That
               (And (Mem ?B U1) (= ?T ?B.Title) (= ?Topic ?B.Topic)))))))


14a: Books-By-Topic Is-A-Member-Of U1db.Reports.
14b: Book Has-A :Property Called "Topic".
14c: Books-By-Topic Is-An-Instance-Of Information-System-Report.
14d: Books-By-Topic Is-An-Instance-Of Tracking-Report.
14e: Books-By-Topic.Accessed-Information Has-Value
     "Topic-of-Book, Title-of-Book".
14f: Books-By-Topic.Target Has-Value U1.
14g: Books-By-Topic.Query Has-Value
     (!Logic (= ?Titles (!The-Set-Of-All (?T %Title) Such-That
      (!There-Exists (?B Book) Such-That
       (And (Mem ?B University-Library) (= ?T ?B.Title)
           (= ?Topic ?B.Topic)))))).
14h: Books-By-Topic.Data-Out Has-Value (!Decl ((?Titles (!Set-Of %Title)))).
14i: Books-By-Topic.Data-In Has-Value (!Decl ((?Topic %Topic))).
```

An interesting aspect of this definition is that it refers to a previously undefined property of a Book, its 'Topic'. The requirements cliché library does not contain any mention of the term topic. However, the context of usage "?B.Topic" (14g) indicates that topic is a property of any arbitrary book. As a result, the RA assumes that topic must be a property of books in general.

This is a limited application of generalizing from instances, representative anonymous instances in particular, to classes. This is a rich area of knowledge acquisition to be explored. For example, after processing this entire scenario the RA will know a great deal about the UL. This knowledge could be generalized to capture a definition of library. Currently, the RA is not used to help populate its own cliché libraries.

Touching New Data

```
15> (Define Books-By-Borrower :Report :Roles
          (:Data-In (!Decl ((?B Borrowers)))
           :Data-Out (!Decl ((?Books (!Set-Of Book))))
           :Query (!Logic (= ?Books (!The-Set-Of-All (?A-Book Book)
              Such-That (Borrower ?A-Book Ul ?B))))))
```

```
15a: Uldb.Item-States :Got-Values Externally-Lost, Located-Out,
       Internally-Lost, Being-Processed, Located-In.
15b: Books-By-Borrower Is-A-Member-Of Uldb.Reports.
15c: Uldb.Target-State :Got-Values Externally-Lost, Located-Out,
       Internally-Lost, Being-Processed, Located-In.
15d: Ul Is-An-Instance-Of Lending-Repository.
15e: Ul.Borrowing-Period Has-Value (!Text "finite").
15f: Ul.Item-States Is-An-Instance-Of Five-State-Model.
15g: Ul.Item-States :Got-Values Externally-Lost, Located-Out,
       Internally-Lost, Being-Processed, Located-In.

15h: Books-By-Borrower Is-An-Instance-Of Information-System-Report.
15i: Books-By-Borrower Is-An-Instance-Of Tracking-Report.
15j: Books-By-Borrower.Accessed-Information Has-Value "none".
15k: Books-By-Borrower.Target Has-Value Ul.
15l: Books-By-Borrower.Query Has-Value
     (!Logic (= ?Books (!The-Set-Of-All (?A-Book Book) Such-That
       (Borrower ?A-Book University-Library ?B)))).
15m: Books-By-Borrower.Data-Out Has-Value (!Decl ((?Books (!Set-Of Book)))).
15n: Books-By-Borrower.Data-In Has-Value (!Decl ((?B Borrowers))).
```

This definition mentions the word borrower which has not been previously used. However, unlike topic, borrower is referred to in the requirements cliché library. In particular, one of its definitions is in conjunction with the cliché lending-repository (a specialization of repository). Understanding the relation borrower results in classifying the UL as a lending repository (15d). When the UL becomes a lending repository it gets a default state model (15f), the five state model. A lending repository's state model describes the states that member objects can be in. The five states are listed in 15g. As a contrast, an alternative state model is the binary model. In this model, objects are either in or out. The state of the UL is tracked by the ULDB, therefore the ULDB's state is the same (15a).

```
New-Information-For-Conflict 1 Regarding-Heuristic-Modification-Of
(User= Remove (Records Check-Out))
Additional-Choices-Exist-For-Substitution
(Return-Lending-Repository Borrow-Lending-Repository)
New-Information-For-Conflict 1 Regarding-Heuristic-Modification-Of
(User= Add (Records Acquisition))
Additional-Choices-Exist-For-Substitution
(Return-Lending-Repository Borrow-Lending-Repository)
```

In the initial response to the first conflict, the analyst asked for suggestions on how to alter two premises. As a result, any new information regarding these suggestions is displayed to the analyst. Since lending repositories support two additional operations not supported by simple repositories, new choices become available.

Maintaining Control of the Process

```
16> (Define Last-Borrower-Of-Book :Report :Roles
          (:Data-In (!Decl ((?A-Book Book)))
           :Data-Out (!Decl ((?B Borrowers)))
           :Query (!Logic (!If (!There-Exists (?A-Borrower)
             Such-That (Borrower ?A-Book Ul ?A-Borrower))
             (= ?B ?A-Borrower) (Last-Borrower ?A-Book Ul ?B)))))

16a: Last-Borrower-Of-Book Is-A-Member-Of Uldb.Reports.
16b: Last-Borrower-Of-Book Is-An-Instance-Of Information-System-Report.
16c: Last-Borrower-Of-Book Is-An-Instance-Of Tracking-Report.
16d: Last-Borrower-Of-Book.Accessed-Information Has-Value "none".
16e: Last-Borrower-Of-Book.Target Has-Value Ul.
16f: Last-Borrower-Of-Book.Query Has-Value
     (!Logic (!If (!There-Exists (?A-Borrower) Such-That
             (Borrower ?A-Book University-Library ?A-Borrower))
       (= ?B ?A-Borrower) (Last-Borrower ?A-Book University-Library ?B))).
16g: Last-Borrower-Of-Book.Data-Out Has-Value (!Decl ((?B Borrowers))).
16h: Last-Borrower-Of-Book.Data-In Has-Value (!Decl ((?A-Book Book))).
```

Prompted with the additional information above, the analyst decides there is now enough information to resolve the conflict. But first, he defines one more report maintaining control of the interaction and his train of thought regarding reports. This is a report definition much like the previous three.

Preparing to Fix the Conflict

```
17> (Go-To-Context Functional-Requirements)

17a: Expecting-Definition-Of Functional-Requirements Of Uldb.
```

The analyst is about to resolve the previous conflict. He moves back to the functional requirements context thinking that he will continue to describe functional requirements after he fixes the problem.

Fixing the Conflict

```
18> (Resolve-Conflict 1)

Conflict #1
Colliding-Definitions-Of Acquisition And ''Check-In''
Both-Are-Realized-As-Equivalent-Instances-Of Action-Tracking-Operation
Explanation:
1.1 (Equal-Defs Check-In Acquisition) is True by Modus Ponens from ...
Premises:
1. (= Isbn (Unique-Id (!Type Book)))
2. (User= Add (Records Acquisition))
3. (User= Remove (Records Check-Out))
...

Resolve conflict?(Y or N) Yes.
Would you like to retract a premise?(Y or N) No.
Would-You-Like-Suggestions-For-Alternate-Premises
 (0 is no, # or list of #s) 3
In-The-Premise (User= Remove (Records Check-Out))
You-Could-Make-A-Substitution-For-The-Word Remove
Which-Should-I-Choose as a :Substitute-Value
(Return-Lending-Repository Borrow-Lending-Repository
 Add-Repository) (1 based, 0 for none) 2
Resolved-Conflict# 1
```

The RA redisplays an explanation of the conflict and enters a dialog about ways to fix the problem. The analyst chooses to have Check-Out record borrow instead of remove. Both borrow and remove deplete the inventory in the collection. Remove does it permanently (or until another addition) and borrow does it temporarily until the expected return. The confusion with Check-Out could be hypothetically attributed to the following sort of thinking: "When a patron removes a book from the library he becomes responsible for that book." Both patrons and staff "remove" Books. What gets hidden is the two different senses of remove.

```
18a: Check-Out.Records Has-Value Borrow-Lending-Repository.
18b: Check-Out.Normal-Pre Has-Value Located-In-Five-State-Model.
18c: Check-Out.Normal-Post Has-Value Located-Out-Five-State-Model.
18d: Check-Out.Input Has-Value (!Text "(?Operator ?The-borrower ?Item)").
18e: Check-Out.Pre Has-Value
     (!Text "(And (Book ?Item) (Patrons ?The-borrower))").
18f: Check-Out.Normal-Effects Has-Value (!Text "located-out").
18g: Check-Out.Records-Function Is-An-Instance-Of Borrow-Action.
18h: Check-Out.Records-Function Is-An-Instance-Of
     Borrow-Lending-Repository-Action.

18i: Lost-Value Add-Repository For "Check-In".Records.
18j: Lost-Value (!Text "(Not (Mem-repository ?Item University-library).")
     For "Check-In".Normal-Pre.
18k: Lost-Value (!Text "(Mem-repository ?Item University-library)")
     For "Check-In".Normal-Post.
18l: Lost-Value (!Text "(?Operator ?The-agent ?Item)") For "Check-In".Input.
18m: Lost-Value (!Text "(And (Book ?Item)
                             (Or (Staff ?The-agent) (Patrons ?The-agent)))")
     For "Check-In".Pre.
18n: Old-> "Check-In" No-Longer-Equals New-> Acquisition.
     As-Type Action-Tracking-Operation Conflict# 1.
```

The problem is resolved since Check-In no longer records add. This is due to the change in its inverse Check-Out. Check-Out obtains a new definition based on borrow (18a–18h). Notice that information about Check-In has been lost. In particular, the value of records (18i) has been lost and no replacement deduced. Information loss is one kind of interesting observation that the RA generates. If a problem is fixed by depriving the RA of information necessary to detect it, then the problem has only been hidden. The analyst decides whether information lost should be present.

Restoring Lost Information

```
19> (Define Check-In :Roles (:Records Return))

19a: Check-In.Records Has-Value Return-Lending-Repository.
19b: Check-In.Normal-Pre Has-Value Located-Out-Five-State-Model.
19c: Check-In.Normal-Post Has-Value Being-Processed-Five-State-Model.
19d: Check-In.Input Has-Value (!Text "(?Operator ?The-borrower ?Item)").
19e: Check-In.Pre Has-Value (!Text "( Book ?Item )").
19f: Check-In.Normal-Effects Has-Value (!Text "located-in").
19g: Check-In.Records-Function Is-An-Instance-Of Return-Action.
19h: Check-In.Records-Function Is-An-Instance-Of
     Return-Lending-Repository-Action.
```

The analyst thinks that borrow and return ought to be inverses so he provides the lost information, defining Check-In to record return (19a). The definition of Check-In is propagated (19b–19h), but a contradiction arises.

A single RA command may generate between 1 and 400 or more interactions with the underlying reasoning system. Any of these interactions that declares a truth value could potentially cause a contradiction. The RA records all contradictions that arise in processing a command. It performs a simple analysis of the contradictions, computing the number of premises underlying each one. Interesting premises are defined as those which originate from the analyst and not from the cliché library. The RA chooses the contradiction with the smallest number of interesting premises to work on first.

```
There-Are 1 Pending-Contras
The-Contra-With-Minimum-Premises-Has 57
With-Minimum-Interesting-Premises-Has 19
Going-For-The-One-With-Smallest-Premises
A-Contradiction-From-A Equality Clause-Between->
(Not (= (Subsume-Cond Being-Processed-Five-State-Model
                      Located-In-Five-State-Model)
        (Subsume-Cond (Normal-Post Check-In) Located-In-Five-State-Model)))
...
The-Interesting-Premises-Are
 (Number-In-Parens-Greater-Than-One-Is-Contras-Premise-Is-In)
...
15. (!Default 15 (Five-State-Model (Item-States University-Library)))
...


Would you like to retract a contra premise?(Y or N) No.
Would-You-Like-Suggestions-For-Alternate-Premises
 (0 is no, # or list of #s) 15
Default (!Default 15 (Five-State-Model (Item-States University-Library)))
Can-Be-Retracted With-The-Following-Replacement-Options-For-Supported-Premise
(Tri-State-Model Binary-State-Model Lending-State-Model)
Should I retract default?(Y or N) Yes.
Possible-Replacement-Types-In-Type-Assertion
(:Unknown (Five-State-Model (Item-States University-Library))) Are
Which-Should-I-Choose as a :Type-For-Replacement
(Tri-State-Model Binary-State-Model Lending-State-Model)
 (1 based, 0 for none) 1
Resolved-Contradiction
```

A contradiction consists of a set of nodes that cannot all have the truth values they currently hold. For example, {A:true, (Not A):true} is the simplest contradiction. The RA displays each of the nodes in the contradiction and a list of premises underlying those nodes. When there is more than one contradiction, the information about the number of contradictions a premise is in is useful. Even though only one contradiction is displayed, the RA computes the underlying premises of each contradiction. This enables the RA to order premises by the number of contradictions they appear in. This can focus the analysts search for a solution by allowing him to consider premises that have the most impact.

The analyst decides that the cause of the problem is the five state model of the UL. Under this model, Check-In and Check-Out cannot be inverses since books that are checked in are explicitly modeled as going into a holding state (e.g., waiting to be reshelved). The node in the contradiction refers to the fact that the post-condition of Check-In (being-processed-five-state-model) is not subsumed by the precondition of Check-Out (located-in-five-state-model). The analyst retracts the default premise and chooses a substitute model as guided by the RA. A tri-state model is chosen to model books as either being stored in the library, lent out to someone, or missing (stolen, misshelved). To avoid a net loss of information in the RKB, the RA has applied a premise altering heuristic to the default type declaration. Alternative types are found by searching the cliché library for sibling types.

```
19i: Uldb.Item-States :Got-Values Missing, Lent-Out, Stored.
19j: Uldb.Target-State :Lost-Values Externally-Lost, Located-Out,
      Internally-Lost, Being-Processed, Located-In.
19k: Uldb.Target-State :Got-Values Missing, Lent-Out, Stored.
19l: Uldb.Item-States :Lost-Values Externally-Lost, Located-Out,
     Internally-Lost, Being-Processed, Located-In.
19m: Ul.Item-States Is-An-Instance-Of Tri-State-Model.
19n: Ul.Item-States :Got-Values Missing, Lent-Out, Stored.
19o: Ul.Item-States :Lost-Values Externally-Lost, Located-Out,
      Internally-Lost, Being-Processed, Located-In.

19p: Check-Out.Normal-Post Has-Value Lent-Out-Tri-State-Model.
19q: Check-Out.Normal-Pre Has-Value Stored-Tri-State-Model.
19r: Check-In.Normal-Post Has-Value Stored-Tri-State-Model.
19s: Check-In.Normal-Pre Has-Value Lent-Out-Tri-State-Model.
```

After resolving the conflict, the state model of the UL and ULDB changes from the five state to the tri-state (19i–19o). The pre- and post-conditions of Check-In and Check-Out are altered to conform to the new state models (19p–19s).

Another Transaction

```
20> (Define Unshelf :Roles (:Records Remove))

20a: Unshelf Is-A-Member-Of Uldb.Functional-Requirements.
20b: Unshelf Is-An-Instance-Of Action-Tracking-Operation.
20c: Unshelf.Target Has-Value U1.
20d: Unshelf.Objects Has-Value
     (!The-Set-Of-All (?O) Such-That (= (Isbn ?O) $Input)).
20e: Unshelf.Object-Type Has-Value Book.
20f: Unshelf.Normal-Post Has-Value
     (!Text "(Not (Mem-repository ?Item University-library))").
20g: Unshelf.Normal-Pre Has-Value
     (!Text "(Mem-repository ?Item University-library)").
20h: Unshelf.Pre Has-Value
     (!Text "(And (Book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent)))").
20i: Unshelf.Input Has-Value (!Text "(?Operator ?The-agent ?Item)").
20j: Unshelf.Records Has-Value Remove-Repository.
20k: Unshelf.Records-Function Is-An-Instance-Of Remove-Repository-Action.
```

The analyst continues defining additional transactions. Unshelf is a transaction that records the permanent removal of a book from the library.

Another Collision

```
21> (Define Unshelf-All :Roles (:Records Remove
        :Objects (!The-Set-Of-All (?O) Such-That (= (Isbn ?O) $Input))))

21a: Unshelf-All Is-A-Member-Of Uldb.Functional-Requirements.
21b: Unshelf-All Is-An-Instance-Of Action-Tracking-Operation.
21c: Unshelf-All.Target Has-Value U1.
21d: Unshelf-All.Object-Type Has-Value Book.
21e: Unshelf-All.Normal-Post Has-Value
     (!Text "(Not (Mem-repository ?Item University-library))").
21f: Unshelf-All.Normal-Pre Has-Value
     (!Text "(Mem-repository ?Item University-library)").
21g: Unshelf-All.Pre Has-Value
     (!Text "(And (Book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent)))").
21h: Unshelf-All.Input Has-Value (!Text "(?Operator ?The-agent ?Item)").
21i: Unshelf-All.Records Has-Value Remove-Repository.
21j: Unshelf-All.Objects Has-Value
     (!The-Set-Of-All (?O) Such-That (= (Isbn ?O) $Input)).
21k: Unshelf-All.Records-Function Is-An-Instance-Of
     Remove-Repository-Action.
```

The analyst describes a transaction Unshelf-All that records the removal of every book with a given ISBN number. In this command, the analyst provides a specification for the objects being tracked (21j), which took on its default value in the previous definition.

Again, the RA is able to deduce the basic features of this transaction. The RA also detects a problem. Since the ISBN number is a unique identifier for books, there can never be two books with the same ISBN. As a result, the kind of removal recorded by Unshelf-All is no different from the removal recorded by Unshelf.

```
Conflict #2
Colliding-Definitions-Of Unshelf-All And Unshelf
Both-Are-Realized-As-Equivalent-Instances-Of Action-Tracking-Operation
Explanation:
1.1 (Equal-Defs Unshelf Unshelf-All) is True by Modus Ponens from ...
Resolve conflict?(Y or N) No.
Conflict #2.
Colliding-Definitions-Of Unshelf And Unshelf-All.
Both-Are-Realized-As-Equivalent-Instances-Of Action-Tracking-Operation.
```

On seeing this conflict, the analyst realizes that the commands up to this point reflect a type/token confusion between a physical copy of a book and the logical notion of a book as a class of copies of books with some particular title, author, and ISBN.

By this time, it may seem that the errors made by the analyst in the scenario are a bit implausible. However, most errors look dumb after you have found them. In addition, the type/token confusion above was not merely fabricated as part of this scenario. Although the original problem statement maintains a general distinction between a book and a copy of a book, several parts of the problem statement suffer from type/token confusion and use the terms incorrectly (see figure 2-5 on page 30).

Realigning Terms

```
22> (Reformulate Book As Copy-Of-Book)

(Define Library :Ako Repository :Defaults (:Collection-Type Book))
*Reformulate*>?(Y or N) Yes.
(Define Book :Ako Physical-Object :Member-Roles (Title Author Isbn))
*Reformulate*>?(Y or N) Yes.
(Define Book.Isbn :Integer :Cardinality Single :Unique-Id T)
*Reformulate*>?(Y or N) No.
(Define Books-By-Author ...
*Reformulate*>?(Y or N) No.
(Define Books-By-Topic ...
*Reformulate*>?(Y or N) No.
(Define Books-By-Borrower ...
*Reformulate*>?(Y or N) No.
(Define Last-Borrower-Of-Book ...
*Reformulate*>?(Y or N) Yes.
```

```
Retracting-Reasons  (5 6 16)
Command (Define Library :Ako Repository
                        :Defaults (:Collection-Type Copy-Of-Book))
Command (Define Copy-Of-Book :Ako Physical-Object
                 :Member-Roles (Title Author Isbn))
Command (Define Last-Borrower-Of-Book ...
```

To fix the problem, a new term corresponding to a copy of a book must be introduced. In addition, something has to be done about the fact that only some uses of the term book in the previous commands refer to the concept of a book. The rest refer to the concept of a copy of a book. The RA provides a special command that can assist with this kind of conceptual realignment. The reformulate command displays all of the commands that contain the term to be split and asks the analyst to specify where reformulation should occur. The choices made in reformulation reflect the following considerations. Libraries physically contain copies of books. The initial definition given for a book really applies to copies of books. However, the ISBN is the unique identifier of a book, not a Copy-Of-Book. The reports Books-By-Topic, Books-By-Author, and Books-By-Borrower refer to the non-physical concept of a book. Otherwise, for example, the response to a Books-By-Author query would contain a title repeated ten times if the library contained ten copies of the particular book. Last-Borrower-Of-Book does, however, refer to physical copies of books, since it used to locate and assess responsibility for a particular resource.

The reformulation mechanism has an interesting interaction with the context mechanism. Each statement is made in a certain context, but at the time of reformulation the context may very well have changed. Reprocessing the statement in the wrong context may cause unex[illegible]ted results. More importantly, not reprocessing the statement in the appropriate context may result in deductions not being made that were made previously. The resulting loss of information is deceiving. The solution is to record with each command the context in which it was made. The reformulation mechanism temporarily restores this context before reprocessing the command.

```
22a: U1.Collection-Type Has-Value Copy-Of-Book.
22b: Check-Out.Pre Has-Value
     (!Text "(And (Copy-of-book ?Item) (Patrons ?The-borrower))").
22c: Check-Out.Object-Type Has-Value Copy-Of-Book.
22d: Check-In.Pre Has-Value (!Text "( Copy-of-book ?Item )").
22e: Check-In.Object-Type Has-Value Copy-Of-Book.
22f: Acquisition.Pre Has-Value
     (!Text "(And (Copy-of-book ?Item)
                  (Or (Staff ?The-agent) (Patrons ?The-agent)))").
22g: Acquisition.Object-Type Has-Value Copy-Of-Book.
22h: Last-Borrower-Of-Book.Data-In Has-Value (!Decl ((?A-Book Copy-Of-Book))).
22i: Unshelf.Object-Type Has-Value Copy-Of-Book.
22j: Unshelf.Pre Has-Value
     (!Text "(And (Copy-of-book ?Item)
                  (Or (Staff ?The-agent) (Patrons ?The-agent)))").
22k: Unshelf-All.Object-Type Has-Value Copy-Of-Book.
22l: Unshelf-All.Pre Has-Value
     (!Text "(And (Copy-of-book ?Item)
                  (Or (Staff ?The-agent) (Patrons ?The-agent)))").

22m: Library  Is-A  Environment, Container, Physical-Object, Agent, Repository.
22n: Library  Has-No-Local-Slots.
     And-Inherited-Slots Product-Functions, $Behaviors, $Agents, $Actions,
     State-Of, Item-States, Patrons, Staff, Storage-Location, Collection,
     Collection-Type.
22o: Copy-Of-Book  Is-A  Physical-Object.
22p: Copy-Of-Book  Has-Local-Slots  Isbn, "Author", "Title".
     And-Inherited-Slots  State-Of.
```

Once the analyst has identified the commands to be altered, the RA retracts the old commands and asserts the new ones. The changes that result are presented to the analyst for review. The retraction and assertion is done in an incremental way so that the new state of the requirements knowledge base can be determined with a minimum of rederivation. The underlying reasoning system makes sure that the rederivation is carried out completely and that every aspect of the requirements knowledge base is rechecked for consistency. This kind of comprehensive support for change is a particularly important aspect of the RA.

No information has been lost in the reformulation. Some functions and preconditions simply refer to Copy-Of-Book where they once referred to Book.

Defining Copy-Of-Book

```
26> (Define Copy-Of-Book.Copy-Num :Ako Integer :Cardinality Single)

26a: Copy-Of-Book Had-A :Property Called Copy-Num Defined.
```

The original definition of book was incomplete when considered in terms of its new use as the definition of Copy-Of-Book. Copies have an additional role, the copy number, that distinguishes physically distinct copies of the same book (i.e., the same text, title, and author).

Fixing Up Some Details

```
27> (Fill-Role Copy-Of-Book.Unique-Id With (!Cross Isbn Copy-Num))

27a: Copy-Of-Book.Unique-Id Has-Value (!Cross Isbn Copy-Num).
There-Are 1 Pending-Contras
The-Contra-With-Minimum-Premises-Has 68
With-Minimum-Interesting-Premises-Has 24
Going-For-The-One-With-Smallest-Premises
A-Contradiction-From-A Axiom DU #27 Clause-Between->
(Not (= (Set-Of-Uid (!Cross Isbn Copy-Num))
        (!The-Set-Of-All (?O)
          Such-That (= ((!Cross Isbn Copy-Num) ?O) $Input))))
...
The-Interesting-Premises-Are
 (Number-In-Parens-Greater-Than-One-Is-Contras-Premise-Is-In)
...
11. (!Default 19 (= (Objects Unshelf-All)
                     (Set-Of-Uid (Unique-Id (Object-Type Unshelf-All)))))
...

Would you like to retract a contra premise?(Y or N)  Yes.
Retract premise # (0 Is None and Loops to all)?  11
You-Chose (!Default 19 (= (Objects Unshelf-All)
                            (Set-Of-Uid (Unique-Id (Object-Type Unshelf-All)))))
Resolved-Contradiction
```

The analyst completes the addition of the Copy-Of-Book concept by redefining the unique-id of a copy to be defined by two properties: ISBN and copy-num. This change causes a contradiction. Previously, the objects role of Unshelf-All, which was set by the analyst, was consistent with its default value given an object-type of book. Now that that object-type has become Copy-Of-Book and the unique-id has been redefined, there is a conflict between the value set by the analyst and the default value. In this case, the analyst simply retracts the default value and the contradiction is resolved. The analyst had actually given the correct objects role specification for Unshelf-All and this is what caused the colliding definition between Unshelf and Unshelf-All in the first place. Had the analyst defined Unshelf-All before Unshelf, he might have been suspicious when the definition of Unshelf-All caused no contradictions with the default value for the objects role.

```
27b: Check-Out.Objects Has-Value
     (!The-Set-Of-All (?O) Such-That (= ((!Cross Isbn Copy-Num) ?O) $Input)).
27c: Check-In.Objects Has-Value
     (!The-Set-Of-All (?O) Such-That (= ((!Cross Isbn Copy-Num) ?O) $Input)).
27d: Acquisition.Objects Has-Value
     (!The-Set-Of-All (?O) Such-That (= ((!Cross Isbn Copy-Num) ?O) $Input)).
27e: Unshelf.Objects Has-Value
     (!The-Set-Of-All (?O) Such-That (= ((!Cross Isbn Copy-Num) ?O) $Input)).
Old-> Unshelf No-Longer-Equals New-> Unshelf-All.
As-Type Action-Tracking-Operation Conflict# 2.
```

Resolving the contradiction allows most transactions to take on a new value for the objects role. Unshelf-All keeps its objects role value based on the ISBN as an identifier of a group of Copy-Of-Book that comprise a single book class.

Finishing the Term Split

```
28> (Define Book :Ako Bag :Member-Roles (Title Author Isbn)
           :Defaults (:Element-Type Copy-Of-Book))

28a: Book Is-A Bag.
28b: Book Has-Local-Slots "Topic", "Title", "Author", Isbn.
     And-Inherited-Slots Elements, Element-Type.
```

The analyst decides to capture the relationship between books and copies by stating the books are a kind of bag (collection) of objects. The elements of this collection are of type Copy-Of-Book. This causes yet another contradiction. This flurry of contradictions highlights the importance of conflict detection and correction in acquiring an evolving description. As more pieces of data are provided and changed there are increasingly many contradictions until everything is tied together neatly. Resolving these contradictions leads to modifications to the idea until closure on the process is finally obtained.

```
There-Are 2 Pending-Contras
The-Contra-With-Minimum-Premises-Has 60
With-Minimum-Interesting-Premises-Has 19
Going-For-The-One-With-Smallest-Premises
A-Contradiction-From-A Disjoint Predicates Clause-Between->
(Copy-Of-Book !A-Book%Books-By-Borrower)
(Bag !A-Book%Books-By-Borrower)
The-Interesting-Premises-Are
 (Number-In-Parens-Greater-Than-One-Is-Contras-Premise-Is-In)
...
19. ((2) (!Choice Disambig 15
          (Borrower-Lending-Repository !A-Book%Books-By-Borrower
                   University-Library !B%Books-By-Borrower)))
```

```
Would you like to retract a contra premise?(Y or N)  Yes
Retract premise # (0 Is None)?  19
You-Chose (!Choice :Disambig 15
           (Borrower-Lending-Repository !A-Book%Books-By-Borrower
                     University-Library !B%Books-By-Borrower))
Resolved-Contradiction
```

This contradiction stems from the definition of the borrowing operation. The currently adopted definition (borrowing as associated with a lending repository) results in two contradictory assertions about the object of the borrowing transaction. First, it must be of the type of the repository collection type, i.e., a Copy-Of-Book. Second, it also must be of type bag (a parent of book) because the report definition of Books-By-Borrower was correctly left unaltered and declared that it was concerned about books borrowed by patrons of the library (as opposed to Copy-Of-Books). The definition of bag used here is one that requires elements in the bag to not be bags themselves. Therefore the two types, bag and Copy-Of-Book, are incompatible. The analyst decides to retract the node that justifies the RA's disambiguation of the concept borrower to borrower in the context of lending repositories. The RA will continue to try to disambiguate the concept of borrower. In this case, the previously chosen solution will not be adopted because it is false. If there is no other option, then this disambiguation task will remain on the agenda.

```
28c: Ul Is-An-Instance-Of Grouped-Lending-Repository.
28d: Ul.Group-Type Has-Value Book.
New-Information-For-Conflict 1 Regarding-Heuristic-Modification-Of
(User= Remove (Records Check-Out))
Additional-Choices-Exist-For-Substitution
(Add-Group-Grouped-Repository Remove-Group-Grouped-Repository)
```

There was another option for disambiguating borrower, i.e., as borrower in the context of grouped lending repositories. The RA chooses this option and discovers that the UL is a grouped-lending-repository (28c) and fills in a value for group-type based on its knowledge that books group Copy-Of-Book (28d). Another option for a previously explored conflict is also brought to the analysts attention, though at this point the analyst is satisfied with its resolution.

Remove-Item Authorization

```
29> (Define R1-Remove :Action-Authorization-Policy
      :Roles (:Restricted-Action Remove-Item
              :Restricted-Group Patrons-Agent
              :Authorized-Group Staff-Agent))

Looking-For-Values-Of (Monitor R1-Remove) Of-Type Agent
Which-Should-I-Choose for value of (Monitor R1-Remove)
(University-Library University-Library-Database) (1 based, 0 for none)  2
29a: R1-Remove Is-A-Member-Of Library-System.Needs.
29b: Uldb Is-An-Instance-Of Advisory-System.
29c: R1-Remove Is-An-Instance-Of Action-Authorization-Policy.
29d: R1-Remove.Restricted-Group Has-Value Patrons-Agent-Repository.
29e: R1-Remove.Authorized-Group Has-Value Staff-Agent-Repository.
29f: R1-Remove.Monitor Has-Value Uldb.
29g: R1-Remove.Restricted-Action Has-Value Remove-Item.
```

In this command, the analyst defines a standard kind of policy, an action authorization policy. Policies are things to be enforced. They are not naturally occurring constraints. As part of the plausibility checking that the RA does, it must know what agent is responsible for monitoring each policy. The statement of a policy implicitly entails the determination of an enforcer. The RA searches for entities to fill the monitor role (entities of type agent) and comes up with the UL or the ULDB. The analyst chooses the ULDB. In order for the ULDB to enforce this policy, it must be capable of issuing authorizations. Satisfying this requirement causes the RA to make a further determination regarding the ULDB, i.e., it is an advisory system (29b).

This policy corresponds to half of restriction R1 in figure 2-5 on page 30.

Add-Item Authorization

```
30> (Define R1-Add :Action-Authorization-Policy
      :Roles (:Restricted-Action Add-Item
              :Restricted-Group Patrons-Agent
              :Authorized-Group Staff-Agent))

Looking-For-Values-Of (Monitor R1-Add) Of-Type Agent
Which-Should-I-Choose for value of (Monitor R1-Add)
(University-Library University-Library-Database) (1 based, 0 for none)  2
30a: R1-Add Is-A-Member-Of Library-System.Needs.
30b: R1-Add Is-An-Instance-Of Action-Authorization-Policy.
30c: R1-Add.Restricted-Group Has-Value Patrons-Agent-Repository.
30d: R1-Add.Authorized-Group Has-Value Staff-Agent-Repository.
30e: R1-Add.Monitor Has-Value Uldb.
30f: R1-Add.Restricted-Action Has-Value Add-Item.
```

Another restriction is stated. If the analyst had chosen to have the UL monitor this restriction, then an interesting situation would have arisen. This statement would be equivalent to asserting a new natural law in the UL environment or it could be viewed as a modeling decision, i.e., only worry about environments where these restrictions will (by some unspecified means) hold. In attempting to understand how the UL can monitor activity, i.e., the same line of reasoning that permitted the RA to deduce that the ULDB was an advisory system, agenda items would be posted. The RA only has knowledge that allows systems to monitor policies. But the UL cannot be a system since it is the environment. These agenda items would be left pending representing a gap in the RA's knowledge.

This policy corresponds to the other half of restriction R1 in figure 2-5.

Resource Control Policies

```
31> (Resource-Control-Policy
      (!Text-Need ''A Borrower may borrow no more
                   than <Book-Limit> Books at a time.''))
```

```
31a: (!Text-Need "A Borrower may borrow no more
                  than <Book-Limit> Books at a time.")
     Is-A-Member-Of Library-System.Needs.
```

Resource control policies are another kind of policy, less well understood by the RA than action authorization policies since they are not captured as clichés. They are simply stated as uninterpreted text.

This policy corresponds to requirement G3 in figure 2-5.

Another Resource Policy

```
32> (Resource-Control-Policy
      (!Text-Need ''A Borrower may not borrow two copies
                   of the same book''))
```

```
32a: (!Text-Need "A Borrower may not borrow two copies of the same book")
     Is-A-Member-Of Library-System.Needs.
```

These policies are a kind of need. They will appear in the needs section of the requirements document. The inclusion of these policies, despite the fact that the RA can do little analysis of them, allows the capture and creation of a more readable and comprehensive requirements document. In addition, the RA can keep track of traceability between needs and system requirements, i.e., each need ought be handled by some combination of functional requirements.

This policy corresponds to requirement G4 in figure 2-5.

Privacy Policies

```
33> (Define R2 :Privacy-Policy
      :Roles (:Restricted-Action Last-Borrower-Of-Book
              :Restricted-Group Patrons-Agent
              :Authorized-Group Staff-Agent))

33a: R2 Is-A-Member-Of Library-System.Needs.
33b: R2 Is-An-Instance-Of Privacy-Policy.
33c: R2.Restricted-Group Has-Value Patrons-Agent-Repository.
33d: R2.Authorized-Group Has-Value Staff-Agent-Repository.
33e: R2.Restricted-Action Has-Value Last-Borrower-Of-Book.
```

Privacy policies are a third kind of policy known to the RA. This policy corresponds to restriction R2 in figure 2-5.

Another Privacy Policy

```
34> (Define R3 :Privacy-Policy
      :Roles (:Restricted-Action Books-By-Borrower
              :Restricted-Group Patrons-Agent
              :Authorized-Group (!The-Set-Of-All (?Patron) Such-That
                 (Or (Staff-Agent ?Patron)
                   (= ?Patron
                      (User-Target (Perform ?Patron Books-By-Borrower))))))


34a: R3 Is-A-Member-Of Library-System.Needs.
34b: R3 Is-An-Instance-Of Privacy-Policy.
34c: R3.Restricted-Group Has-Value Patrons-Agent-Repository.
34d: R3.Authorized-Group Has-Value
     (!The-Set-Of-All (?Patron) Such-That
      (Or (Staff-Agent ?Patron)
          (= ?Patron (User-Target (Perform ?Patron Books-By-Borrower))))).
34e: R3.Restricted-Action Has-Value Books-By-Borrower.
```

This policy corresponds to restriction R3 in figure 2-5.

The RA's demonstrated understanding of policies in this scenario is limited. Such understanding requires rather in depth understanding of the statements made, often relying on knowledge of first principles involved with the policy. For example, consider a privacy policy and some of the questions that might need to be answered about it. What is privacy? How do privacy goals interacts with other goals and how are tradeoffs to be made? How can a functional requirement be evaluated with respect to supporting a goal, conflicting with a goal, or being irrelevant to a goal? Fickas and Robinson [29, 30, 31] at the University of Oregon have more directly addressed these concerns. Indicative of the difficulty of the problem is the fact that in their KATE project "correspondence links were forged between components in the model and those of the example" [31], i.e., the links were installed manually. The RA is capable of resolving informalities regarding such policies, but it too would require manual installation of relationships among statements in the analysis dialog and the cliché library. This is inconsistent with one of the goals of the RA project which is to push the limits of knowledge reuse in requirements acquisition.

Data Constraints

```
35> (Go-To-Context Data-Constraints)

35a: Expecting-Definition-Of Data-Constraints Of Uldb.
```

Data constraints are constraints on the ULDB information system. Unlike constraints on the UL, the RA has the power to make changes to the system to ensure that these constraints will be met.

A Model Conflict

```
36> (!For-All* (?Record) It-Is-True
      (Not (And (Available ?Record Uldb) (Checked-Out ?Record Uldb))))

36a: Uldb.Item-States Is-An-Instance-Of Binary-State-Model.
36b: Ul.Item-States Is-An-Instance-Of Binary-State-Model.
```

This constraint corresponds to requirement G2 in figure 2-5. This statement expresses a constraint on the ULDB, but since the ULDB tracks the UL, they are constrained to have the same state model. This constraint implies a binary state model for the ULDB (36a) which is inconsistent with the UL's current tri-state model.

```
There-Are 9 Pending-Contras
The-Contra-With-Minimum-Premises-Has 59
With-Minimum-Interesting-Premises-Has 20
Going-For-The-One-With-Smallest-Premises
A-Contradiction-From-A Modus Ponens Clause-Between-> ...
The-Interesting-Premises-Are
 (Number-In-Parens-Greater-Than-One-Is-Contras-Premise-Is-In)
...
8. ((9) (!Default 2 (= (Item-States University-Library-Database)
                       (Item-States (Target University-Library-Database)))))
...
Would you like to retract a contra premise?(Y or N) Yes.
Retract premise # (0 Is None)? 8
You-Chose (!Default 2 (= (Item-States University-Library-Database)
                         (Item-States (Target University-Library-Database))))
Resolved-Contradiction
```

Nine contradictions are caused by this statement, but there is a premise underlying all of them. They can be corrected by retracting the default assumption that a tracking system and its target must have the same state model. The ULDB has a binary state model and the UL has a tri-state model. The fact that they do not have the same model is indicative of the fact that the tracking system has made a simplification regarding how it is modeling the library. This simplification is similar to that made by the MIT library database BARTON [32], i.e., "When an item's status reads 'in library' it means 'not checked out'. If you can't find it on the shelf, ask a librarian." In other words, the system does not account for the possibility of shelving delays, misshelving, or theft. This simplification introduces a validation concern (is such a simplification tolerable) and introduces a need for some sort of inventory procedure to resychronize the state of the system with the actual state of the system.

```
36c: Uldb.Item-States :Got-Values Available, Checked-Out.
36d: Uldb.Item-States :Lost-Values Missing, Lent-Out, Stored.
36e: Ul.Item-States Is-No-Longer-An-Instance-Of Binary-State-Model.
```

The RA captures an understanding of the library problem as it processes the 36 statements that comprise the library scenario. This understanding is possible due to the existence of an appropriate domain model and algorithms that apply relevant information in understanding the evolving description. The next section illustrates excerpts from the requirements document produced after the conclusion of this scenario.

2.9 Library-System Requirements Document Excerpts

This section contains three excerpts from the requirements document produced by the RA after processing the library scenario. Note, however, that the analyst can ask for the document to be produced at any point in the acquisition. The complete document is included in Appendix B. It is supplemented with a reader's guide (in Appendix A) that describes the conventions used in producing the requirements document.

One obvious property of the document is its large size compared to the interactive input that produced it. The IEEE library problem statement is one page long. The 36 commands to the RA are two to three pages long. The resulting requirements document is 24 pages long. Part of this expansion is attributable to the organizational structure of the document, e.g., a minimum RA requirements document is five pages long (a page for each chapter). Incorporation of information from the cliché library is also responsible for a large part of the expansion.

The large size of the document is not, a priori, a positive or negative feature. Requirements documents are typically large because they need to describe a problem and a system in precise detail. In the RA, the length added to the document by a statement is proportional to its information content. In more realistic requirements, much of the document is devoted to relatively low information content, highly redundant (between two requirements) specifications of, for example, screen output formats or transaction I/O definitions. Thus, the length of the RA's requirements document remains proportional to the information content of the input and should not expand prohibitively.

Before continuing with the description of the selected sections, the reader may want to turn to the appendix and look at the requirements document table of contents. It provides an outline and reasonably good overview of the document. In particular, notice the enumeration of various features of the requirement, e.g., the functional requirements as subsections of document section 4.1 and the product functions as subsections of document section 3.2.

(The rest of this section consists of a page from the requirements document facing a page of commentary.)

1 Introduction

This is the library-system requirement. The problem is set in an environment described as the university-library (UL). The university-library-database (ULDB) provides the solution.

UL is a grouped-lending-repository.

The UL stores copy-of-books organized as groups of books. Copy-of-books that are in the repository may be borrowed for a finite period of time by the patrons. The repository's collection may shrink or grow in size as it is maintained by the staff.

ULDB is an advisory-system and tracking-information-system.

The ULDB regulates the ULDB.system-monitored (unknown) using notifications and authorizations to ULDB.advice-recipients (unknown). The ULDB monitors and attempts to enforce a set of restrictions placed on a target physical system. The enforcement power of the ULDB is, however, purely advisory.

The ULDB assists the patrons and staff of the UL by tracking copy-of-books. It maintains an internal image of the changing physical state of the UL. Reports may be generated from this internal image to summarize the state of the UL.

There are 12 statements that define the system needs. They are divided into 3 categories: polices (6 statements), product functions (5), and needs (1). The following more specialized need types are used in the definition of some of the 12 needs statements: action-authorization-policy, privacy-policy, and resource-control-policy.

See the accompanying reader's guide for assistance in interpreting this document.

Figure 2-8: Document Introduction

The document's introduction provides a high level overview of the entire requirement. Its organization closely follows that of the complete document. The first paragraph is an overview of the introduction that identifies the requirement by naming it and its central agents. Next, a high-level description of the environment is provided, followed by a high-level description of the system. These descriptions are generated by filling in the abstract level canned text for the appropriate clichés. Following this, the needs are summarized by an inventory that lists their breakdown into categories. A concise summary of needs is difficult since they are stated in an eclectic fashion.

The introduction is generated automatically by the RA using the appropriate instances from the RKB and the appropriate canned text from the cliché library. A major advantage to automatic production is that all parts of the document are synchronized with the current state of the requirements analysis. In particular, a high-level introductory description, which is derivative from the main body of the requirement, is a likely candidate for falling out of synch since refinements are generally made to the body of the requirement versus the introduction in order to move the analysis forward.

The last sentence of the introduction points to the reader's guide which explains a number of document reading conventions, e.g., that role designators (e.g, obj.role) followed by the word unknown in parenthesis indicate roles that the document generator tried to access but that were unfilled.

2 Environment

The university-library (UL) is a grouped-lending-repository.

The UL is a repository of objects of type copy-of-book. These objects are grouped into classes of book in which individual objects are considered interchangeable. The staff of the repository regulates the collection by deciding when objects in the collection need to be removed or added. The patrons of the repository borrow and return items in the collection.

The roles of the UL are:

- Collection-type: copy-of-book.

  is the type of physical entities in the collection.

- Group-type: book.

  is the type of abstract entities into which collection members are grouped.

- Item-states: lent-out, missing, and stored.

  describes the possible states that component items may be in.

  Copy-of-book that are members of the UL may be in one of three states: stored, lent-out, and missing. This trinary model is based on the assumption that items to be lent may be in, out, or unaccounted for. This model does not make distinctions between the ways in which an item may be unaccounted for.

- Access: unknown.

  describes limits on the execution of certain actions by certain privileged or non-privileged groups.

- Borrowing-period: finite.

The unfilled slots are state-of.

Book is a type. Its parent types are bag. Book has local slots author, isbn, title, and topic. Book inherits 2 other slots from its parents. Book is a group of copy-of-book.

Copy-of-book is a type. Its parent types are physical-object. Copy-of-book has local slots author, copy-num, isbn, and title. Copy-of-book inherits 1 other slots from its parents.

Library is a type. Its parent types are repository. Library has no local slots. Library inherits 11 other slots from its parents.

Figure 2-9: Environment Description

The second excerpt is a description of the requirement's environment. The first part of the description describes the UL as a member of the most specific cliché(s) it is known to be an instance of (in this case grouped lending repository) by using the appropriate overview level canned text. This is followed by an enumeration of the roles applicable to the UL. Each role is described by listing its value (if known), printing any static role documentation that describes the purpose of the role, and describing it as an instance of the applicable clichés. A list of the unfilled roles follows the role enumeration. Notice also that a portion of the dictionary is factored out and included after the environment description. New terms accessed by the environment description are defined immediately following it to assist in the flow of the section.

All the information known about the UL is summarized in this section of the document. One major feature of the document is that it provides a global, integrated view of the requirement (as compared, for example, to the immediate feedback). The full description of the UL was acquired gradually throughout the scenario. The summarization here facilitates review of the aggregate description acquired.

4.1 Functional-Requirements of ULDB

The functional requirements of a System are a set of operations that the system must support in order to satisfy the defined needs of the system. For purposes of validation functional requirements should be traceable to specific needs and every need should be accounted for by one or more functional requirements

4.1.1 Acquisition

Acquisition is an action-tracking-operation.

Acquisition records the add-repository of copy-of-book objects from UL by acquisition.the-agent (unknown).

- input: (?Operator ?The-agent ?Item).
- pre: (And (Copy-of-book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent))).
- normal-pre: (Not (Mem-repository ?Item UL)).
- normal-post: (Mem-repository ?Item UL).
- records: add-repository.
- target: UL.
- object-type: copy-of-book.
- objects: (!the-set-of-all (?o) s.t. (= ((!cross isbn copy-num) ?o) input)).

The unfilled slots are post and normal-effects.

4.1.2 Check-in

Check-in is an action-tracking-operation.

Check-in records the return-lending-repository of copy-of-book objects from UL by check-in.the-agent (unknown).

- input: (?Operator ?The-borrower ?Item).
- pre: (Copy-of-book ?Item).
- normal-pre: lent-out-tri-state-model.
- normal-effects: located-in.
- normal-post: stored-tri-state-model.
- records: return-lending-repository.
- target: UL.
- object-type: copy-of-book.
- objects: (!the-set-of-all (?o) s.t. (= ((!cross isbn copy-num) ?o) input)).

The unfilled slots are post.

Figure 2-10: Functional Requirements Section

The third excerpt is the beginning of the section that enumerates the functional requirements of the ULDB. These requirements are grouped together in this section and share a common format as dictated by the sharing of many different roles, e.g., input, normal-pre, and normal-post.

Each functional requirement is described in summary as an instance of the applicable cliché(s) followed by an enumeration of its filled roles. The roles are instantiated with information specific to this particular requirement, e.g., the pre(condition) of acquisition mentions copy-of-book, staff, and patrons.

2.10 The Air Traffic Control Scenario

This section contains a trace of the RA acquiring a requirement in the air traffic control domain, thus illustrating the range of applicability of the RA. The key features of the following scenario are simply that it shares the use of several of the clichés used in the library scenario and that exactly the same reasoning, disambiguation, and conflict detection mechanisms are used. To emphasize this, the description of the scenario is kept at a more conceptual level. The scenario is described in detail (along with a description of the user's experience with the RA) in [28].

As an *example of a cliché used* in both scenarios, consider that the radar defined in command 3 of figure 2-11 is a tracking system. The radar tracks many targets rather than just one, operates by direct observation (command 4) within a restricted volume rather than by indirect observation, and is not an information system. Nevertheless, the radar shares with the ULDB the key characteristic of tracking the state of other objects.

As an example of a different cliché, consider that the radar-display defined in command 6 is a display system, a system that maintains a continually updated display of some quantity. In addition, the commands in the air traffic control scenario make use of a number of clichés involving different kinds of *volumes and errors*. The radar observes aircraft within a limited cylindrical volume and has an observational error that is naturally characterizable in polar coordinates.

The aircraft tracking scenario is presented in the following 6 figures.

```
1: (Find-Requirement Aircraft-Tracking-System)
...
2: (Define Aircraft :Ako Physical-Object
     :Member-Roles ((The-Trajectory Trajectory) (Flight-Plan Trajectory)))
Aircraft Is-A Physical-Object.
...
3: (Define Radar :Tracking-System :Roles (:Target (!Set-Of* Aircraft)))
Radar Is-An-Instance-Of Multiple-Target-Tracking-System.
...
4: (Fill-Roles Radar.Observation-Region
     :Max-Radius (Miles 50) :Max-Altitude (Feet 40000))
Radar Is-An-Instance-Of Multiple-Target-Direct-Observation-Tracking-System.
Radar.Observation-Region Is-An-Instance-Of Cylindrical-Volume.
...
5: (Fill-Roles Radar.Observation.The-Error
    :Range-Error (Feet 20) :Theta-Error (Degrees 1)
    :Azimuth-Error (Degrees 1))
Radar.Observation-Region Is-An-Instance-Of Cylindrical-Polar-Volume.
Radar.Observation-Region.Max-Azimuth Has-Value (!Radians 1.57).
Radar.Observation.The-Error Is-An-Instance-Of Limited-Polar-Error.
...
```

Figure 2-11: The Aircraft Tracking System

In figure 2-11, the analyst begins a requirement for an aircraft tracking system (command 1). As in the library scenario, the analyst starts by introducing some basic entities relevant to the requirement: aircraft (command 2) is the kind of entity to be tracked, and the radar (command 3) is the tracking system. Command 3 says that the target of the radar tracking system will be an unspecified set of aircraft. Since the radar is tracking a set of objects, the RA deduces that the radar system is a multiple target tracking system.

In command 4, the analyst describes the size of the region that the radar will be responsible for monitoring. The analyst defines the radius and altitude of the region to be covered (leaving out a specification of the region's center), thus defining a cylindrical volume of responsibility. The RA deduces further that the radar is a direct observation tracking system because its range of observation is limited. Command 5 states the accuracies of the radar's observations in polar coordinate units. This causes the RA to perform a coercion between the cylindrical units specified as the regions size and the polar tolerance. Clichés have been defined that capture the notion of coercion between types. The RA applies this information by deducing that the region is an instance of the cylindrical polar volume coercion cliché. The coercion permits computing a number of polar coordinate values for the region including maximum azimuth and range. The tolerance specification is for errors within a limited region and therefore the error is classified as a limited polar error. This knowledge is useful because in an infinite region the magnitude of an error may increase as distance from the radar increases.

As in the library scenario, the first few commands simply set the stage and build up a critical mass of facts to permit more in-depth analysis.

```
6: (Define Radar-Display :Display-System
                         :Roles (Display-Of Radar.Observation))
...
7: (Fill-Roles Radar-Display.Display.The-Error
    :Latitude-Error (Feet 300) :Longitude-Error (Feet 300)
    :Altitude-Error (Feet 300))
Radar.Observation.The-Error Is-An-Instance-Of Limited-Polar-Cartesian-Error.
Radar.Observation.The-Error.Altitude-Error Has-Value (!Meters 1978.84).
Radar-Display Is-An-Instance-Of With-Error-Display-System.
Radar-Display.Display.The-Error
                        Is-An-Instance-Of Limited-Polar-Cartesian-Error.
Radar-Display.Display.The-Error.Altitude-Error Has-Value (!Meters 93.0).
...
There-Are 1 Pending-Contras
A-Contradiction-From-A Modus Ponens Forward Clause-Between->
(Not (< (Number-Of (Altitude-Error Radar.Observation.The-Error)) 93.0))
(< (Number-Of (Altitude-Error Radar.Observation.The-Error)) 93.0))
...
Would you like to retract a contra premise?(Y or N) Yes.
Retract premise # (0 Is None and Loops to all)? 13
You-Chose (!Default 8 (= (Altitude-Error Radar.Observation.The-Error)
               (Meters (Rat->Alt-Acc
                 (Number-Of (Range-Error Radar.Observation.The-Error))
                 ....))))
Resolved-Contradiction
Lost-Value (!Meters 1978.84) For Radar.Observation.The-Error.Altitude-Error.
8: (Define Beacon :Data-Source
                  :Roles (:Error-Value (Feet 100) :From Aircraft))
...
9: (Fill-Role Radar.Observation.The-Error.Altitude-Error
              With Beacon.Error-Value)
Radar.Observation.The-Error.Altitude-Error Has-Value (!Meters 31.0).
```

Figure 2-12: Incompatible Accuracy Requirements

In command 6, another component of the system is introduced, the display that will present the radar's observations. In command 7, the analyst specifies that the positions of aircraft must be displayed with an error of no more than 300 feet. The error tolerance specified for the display is in latitude and longitude, forcing yet another coercion, from cartesian to polar coordinates. Each requirement is stated in units natural to the function being described. The RA handles the chore of translating between the various measures. The specified error tolerance leads immediately to a conflict. One of the constraints on the display system cliché specifies that the accuracy of the display is inherently limited by the accuracy of the data. Here, it is impossible to achieve the required altitude accuracy in the display given the long range and large azimuth error of the radar. One either has to use an improved radar or settle for less display accuracy.

The analyst knows that there is a more accurate source of aircraft altitude data. In preparation for this addition, he retracts the default formula which states how altitude error is computed from the size of the region and the other error tolerances. Command 8 introduces a new more accurate aircraft altitude measure that is known to be provided by an aircraft transponder beacon. This new altitude value is consistent with the desired display tolerances. Command 9 defines the beacon as the alternate source for radar altitude data.

```
10: (Define Aircraft-Monitoring-System :System :Synonym AMS)
...
11: (Need (!Logic (Monitors AMS (!Image-Of (!Set-Of* Aircraft) Trajectory)
                               (!Image-Of (!Set-Of* Aircraft) Flight-Plan))))
AMS Is-An-Instance-Of Multi-Target-Monitoring-System.
AMS.Predicted Has-Value (!Image-Of (!Set-Of* Aircraft) Flight-Plan).
AMS.Actual Has-Value (!Image-Of (!Set-Of* Aircraft) Trajectory).
...
12: (Fill-Role AMS.Observation-Region With Radar.Observation-Region))
Radar.Observation-Region Is-An-Instance-Of Cylindrical-Trajectory-Volume.
Radar.Observation Is-An-Instance-Of Type-With-Limited-Cylindrical-Position.
AMS Is-An-Instance-Of Multiple-Target-Direct-Observation-Monitoring-System.
...
There-Are 14 Pending-Contras
Would you like to retract a contra premise?(Y or N) Yes.
Retract premise # (0 Is None and Loops to all)? 7
You-Chose (User= (Observation-Region Radar)
                 (Observation-Region Aircraft-Monitoring-System))
Resolved-Contradiction
...
Radar.Observation-Region Is-No-Longer-An-Instance-Of
                         Cylindrical-Trajectory-Volume.
AMS Is-No-Longer-An-Instance-Of
    Multiple-Target-Direct-Observation-Monitoring-System.
...
13: (Fill-Role AMS.Observation-Region.Object-Volume
               With Radar.Observation-region))
AMS Is-An-Instance-Of Multiple-Target-Direct-Observation-Monitoring-System.
Aircraft-Monitoring-System.Observation-Region
         Is-An-Instance-Of Trajectory-Volume.
```

Figure 2-13: The Aircraft Monitoring System

Command 10 introduces the aircraft monitoring system, which, in command 11, is defined to monitor aircraft trajectories for their compliance with aircraft flight plans (`!image-of` returns the image set (range) of a property accessor applied to a set).

Command 12 connects the AMS with the radar system by saying they observe the same region in space. This causes a contradiction because the AMS deals with trajectories (a historical sequence of positions) and the radar deals with positions. This causes the region to be modeled both as a trajectory volume and a position volume which are disjoint types, a subtle modeling error. However, a trajectory volume does have a property, object volume, which denotes the actual region in which the trajectories are being generated.

The analyst handles the contradiction in command 12 by first retracting the effect of the command. In command 13, the analyst asserts the proper relationship by saying that the object-volume of the AMS observation-region is the same physical region as that observed by the radar.

```
14: (Define (Add-Traj Add-Fp Aircraft-Hand-In Aircraft-Take-Off) :Event)
Add-Traj Is-An-Instance-Of Event.
...
15: (Need (!Logic (Preceded-By-Strictly Add-Traj
                          (One-Of Aircraft-Hand-In Aircraft-Take-Off))))
...
16: (Need (!Logic (Comes-Before Add-Fp
                       (One-Of Aircraft-Hand-In Aircraft-Take-Off))))
...
17: (Need (!Logic (Error-When (And (Event-Happens Add-Traj)
                                   (Not (Event-Happens Add-Fp))))))
...
```

Figure 2-14: Ordering Typical Events

The analyst now introduces some typical events, in command 14, that occur in the course of normal aircraft monitoring. Add-Traj is an add trajectory event corresponding to the AMS observing a new trajectory. Add-Fp is an add flight plan event that is also processed by the AMS. Aircraft-Hand-In and Aircraft-Take-Off are two events that happen in the world of aircraft that bear special relationship to adding trajectories and flight plans.

Command 15 asserts the need that before trajectories are added either a hand-in or a take-off should happen, i.e., the two ways that aircraft come into existence in an AMS are either by being handed-in from another AMS or by taking off in the current region.

Command 16 states that before an aircraft physically enters an airspace that it should be preceded by a flight-plan. Command 17 states that it is an error if the AMS observes an unaccounted for aircraft, i.e., a trajectory with no associated flight plan.

The analyst has introduced these representative events in order to assert some correctness conditions. In some sense, the above statements are actually quantified over all events in that class. The analyst, is, in effect, using the RA to specify typical test scenarios in order to make sure that the conditions specified are sensible (e.g., mutually consistent).

There is actually a problem with command 17 which will crop up later in the scenario. For now, the user continues to define some more correctness conditions.

```
18: (Need (!Logic (Error-When
     (!There-Exists ((?Aircraft1 Aircraft) (?Aircraft2 Aircraft)) Such-That
       (< (Distance ?Aircraft1.Trajectory ?Aircraft2.Trajectory)
          (Feet 400))))))
...
19: (Need (!Logic (Error-When
      (!There-Exists ((?Aircraft1 Aircraft) (?Aircraft2 Aircraft)) Such-That
       (< (Distance (Flight-Plan ?Aircraft1) (Flight-Plan ?Aircraft2))
          (Feet 700))))))
...
20: (Need (!Logic (Error-When
      (!There-Exists (?Aircraft1 Aircraft) Such-That
        (< (Distance ?Aircraft1.Trajectory (Flight-Plan ?Aircraft1))
           (Meters 300))))))
...
21: (Think-Harder)
Conflict-# 2
Error-When (Not (And (Event-Happens Add-Traj) (Not (Event-Happens Add-Fp))))
Has-Static-Truth-Value-Is-Tautologous
Explanation: ...
Premises: ...
Would you like to retract a premise?(Y or N) Yes.
Retract premise # (0 is None)? 2
You-Chose (Need (!Logic (Preceded-By-Strictly Add-Traj
                         (One-Of Aircraft-Hand-In Aircraft-Take-Off))))
Resolved-Conflict# 2
...
22: (Need (!Logic (Preceded-By-Strictly Add-Traj
     (One-Of Aircraft-Unauthorized-Entry Aircraft-Hand-In
             Aircraft-Take-Off))))
...
23: (Think-Harder)
```

Figure 2-15: Thinking Harder

Commands 18, 19, and 20 state some more error conditions on the system. Command 18 says it is an error if two aircraft get too close together. Command 19 says it is an error if two flight plans are assigned too close together. Command 20 says it is an error if an aircraft strays too far from its flight plan.

The analyst finishes off the conditions he is working on and decides to take a break. While he is away, he instructs the RA to think harder (command 21). Most reasoning systems need to tradeoff completeness for execution speed. In a system such as the RA, interactive response time is important and therefore the tradeoff will need to be made in favor of quicker execution. The think-harder command sacrifices response time and tells the system to reason more completely. In this case, the RA observes that the error condition stated in command 17 can never occur and thus is redundant. This may be indicative of a misconception, since presumably the analyst thought the condition could happen if he discussed it.

In command 21, the analyst retracts the source of the problem, i.e., that only hand-ins and take-offs can cause the addition of a trajectory. Command 22 adds the possibility of an unauthorized aircraft entry. The analyst rechecks the situation by asking the RA to think harder, in command 23, and nothing new is found.

```
24: (Need (!Logic (Match (!Set-Of* Aircraft) (Observation Radar))))
Need (Match (Observation Radar) (!Set-Of* Aircraft))
Is-Statically-True-In-Current-Description.
Explanation: ...
25: (Reformulate Aircraft As Blip)
...
(Define Radar :Tracking-System :Roles (:Target (!Set-Of* Aircraft)))
=Reformulate=>?(Y Or N) Yes.
(Define Beacon :Data-Source :Roles (:Error-Value (Feet 100) :From Aircraft))
=Reformulate=>?(Y or N) No.
(Need (!Logic (Monitors Aircraft-Monitoring-System
                        (!Image-Of (!Set-Of* Aircraft) Trajectory)
                        (!Image-Of (!Set-Of* Aircraft) Flight-Plan))))
=Reformulate=>?(Y Or N) No.
(Need (!Logic (Error-When
       (!There-Exists ((?Aircraft1 Aircraft) (?Aircraft2 Aircraft)) Such-That
        (< (Distance ?Aircraft1.Trajectory ?Aircraft2.Trajectory)
           (Feet 400))))))
=Reformulate=>?(Y Or N) Yes.
...
(Need (!Logic (Error-When
        (!There-Exists (?Aircraft1 Aircraft) Such-That
         (< (Distance ?Aircraft1.Trajectory (Flight-Plan ?Aircraft1))
            (Meters 300))))))
=Reformulate=>?(Y Or N) No.
Command  (Define Radar :Tracking-System :Roles (:Target (!Set-Of* Blip)))
Command (Need (!Logic (Error-When
         (!There-Exists ((?Aircraft1 Blip) (?Aircraft2 Blip)) Such-That
          (< (Distance ?Aircraft1.Trajectory ?Aircraft2.Trajectory)
             (Feet 400))))))
...
Radar.Target Has-Value (!Set-Of* Blip).
```

Figure 2-16: Another Conceptual Error

Up until this point, the analyst has not actually said anything that ties the set of aircraft observed by the radar system to the trajectories and flight plans monitored by the AMS. In command 24, the analyst rectifies this by asserting that the observations made by the radar need to be matched (i.e., identified in a one-to-one mapping) with the set of aircraft being observed. In this way, the aircraft can be identified by the tracking system and then further matched to the data stored by the AMS on aircraft trajectories and flight-plans. The RA reports that this matching condition is already statically true. This surprises the analyst because he did not think he said enough to deduce this. The problem is that the analyst has left unstated an extra level of modeling that he was implicitly assuming.

In command 25, the analyst makes this extra level of modeling explicit. The radar tracking system is actually provided its data by the radar sensor which outputs radar blips. Another mechanism is responsible for deciding that a sequence of blips over a period of time correspond to the track of a particular aircraft.

The RA assists the analyst in debugging the aircraft tracking system description by detecting contradictions and other unusual circumstances. It applies knowledge from the cliché library in order to assist in the elaboration of the description, performing coercions and classifying the various system components as more information is discovered about them. Reformulation is again used to clear up one of the conceptual confusions. The same capabilities used in the library scenario prove useful in formulating this scenario.

Chapter 3

Principles of Operation

This section explains how the Listener operates and how it accomplishes the behavior demonstrated in the preceding sections and in Section 4. The explanation is presented from three viewpoints. First, the capabilities of the Listener are explored from the speaker's point of view. This provides operational expectations for the users of the Listener system. Second, the techniques used to support the capabilities are discussed. This provides a guide to would-be designers of similar systems. Third, some of the implementation details are described. This provides assistance to someone trying to replicate this effort.

3.1 Capabilities

From the speaker's viewpoint, the capabilities of the Listener can be grouped into the five categories listed below. As described in the following subsections, this suite of capabilities permits the speaker to interact effectively with the Listener.

- clichés
- informality resolution
- support for evolution
- incremental feedback
- summarization

Clichés are the heart of the language shared by the speaker and the Listener, providing a common terminology for the transfer of knowledge between them. Clichés are valuable, in part, because they provide an effective way to communicate.

Informality resolution allows the Listener to fill in details implicit in the speaker's exposition. This allows a more efficient conversation since the speaker is not forced to describe everything in tedious detail.

No matter how efficient and effective the communication channel is made, provision must be made for errors (or simply changes) in the conversation. Support for evolutionary conversational techniques permits the speaker to alter previous statements without requiring a replay of the entire conversation.

Incremental feedback provides an evolving picture of the state of the conversation. It is assumed that the speaker will digest the feedback and that this will either: trigger new lines of thought, highlight problems in the current description, or confirm the current line of thought.

The above four capabilities address the needs of a conversational interaction between speaker and listener. Summarization (the generation of a summary document) supports the more contemplative process of an editing cycle between author and reviewer. This capability is not necessary for statement by statement understanding; however, it provides a comprehensive, global view of the results of the description acquisition, complementing incremental feedback.

3.1.1 Clichés

Clichés are the concepts shared by the speaker and the Listener. As such, clichés are the units of knowledge reuse. A cliché consists of a set of roles and a set of constraints. Roles are placeholders to be filled by other objects. Constraints relate the roles. The choices made in filling the roles determine a particular instantiation of the cliché.

For example, the concept *add-object-to-collection* (illustrated below) is a cliché that can be loosely defined as follows: An *object* is added to a discrete *collection* by some *agent*. The operation is restricted to objects of an appropriate type for the collection and agents that are authorized to perform the addition. The precondition of the operation requires that the object must not already be in the collection, and the postcondition states that the object becomes a member of the collection.

```
add-object-to-collection
 roles: object, collection, agent
   constraints:
     (discrete-collection collection)
     (= (type-of object) (element-type collection))
     (authorized-to-add agent collection)
     (precondition (not (member object collection)))
     (postcondition (member object collection))
```

For the statements exchanged between the speaker and the Listener to have meaning, at least some of the words used in those statements must have meaning to the

Listener. Clichés provide this meaning. The speaker builds new ideas from a combination of (partially) instantiated clichés and logical assertions[1].

The utility of using a cliché in conversation is that the use of a single word imports the cluster of information that defines the cliché, i.e., roles and constraints. This utility increases with the number of clichés known to the Listener and shared with the speaker. The Listener can make more, and deeper, deductions given greater common ground with the speaker. To provide substantial common ground, the Listener must be provided with an extensive cliché library relevant to the idea under discussion.

```
add (?object ?destination ?agent)
 |
 |--add-numbers
     |--add-complex-numbers
     |-- ...
 |--add-object-to-collection
 |--augment-continuous-quantity
 |--join-collections
 |-- ...
```

Figure 3-1: Hierarchy of Clichés

Figure 3-1 shows a fragmentary skeleton of a cliché library. It is organized in a specialization hierarchy. Each child is a refined version of its parent, generally containing more detailed and more specific information. *Add* is an operation with three roles, whose relationship is loosely defined as follows: an *object* is added to some *destination* by some *agent*. *Add* is specialized in a number of ways:

- *add-numbers*: e.g., Scott added 3 to 4.
- *add-complex-numbers*: e.g., Denise added 4-5i to -7+3i.
- *add-object-to-collection*: e.g., Mark added a quarter to the pot.
- *augment-continuous-quantity*: e.g., Lisa added a few pounds (over vacation).
- *join-collections*: e.g., The AFL-CIO added their support to the teacher's strike.

The cost of constructing a cliché library can be amortized over its reuse in many description acquisitions in the same general topic area. Candidate domains for reuse should have a rich, relatively stable, vocabulary of technical terms that are understood well enough to be codified as clichés.

[1]For example, see command 5 on page 35 where *library* is defined as a *repository* of *books*.

The semantics of a cliché are enriched by the reasoning mechanisms that operate on the data represented in the cliché. These mechanisms are outlined in the next section. In addition, the general theory of clichés only takes on significance when a set of concepts is actually identified and codified. This has been partially illustrated with respect to the Library scenario and is discussed in greater detail in Section 3.2.2 which explores cliché representation issues.

3.1.2 Informality Resolution

Informality is an inevitable property of the speaker's description to the Listener. To explain what is meant by informality, an operational definition is provided. Then, a structure for understanding the set of informalities as a whole is presented. Finally, given an understanding of informality, the need for (and process of) informality resolution is explained.

The speaker may make use of the following informal techniques [33, 1] in conveying information to the Listener.

- *Ambiguity* - A statement is ambiguous when the meaning it conveys is insufficient to permit choosing the most specific interpretation. Since this choice depends on how far along in the description the speaker is, many statements will be ambiguous until the speaker reaches a point quite far along in the description.
- *Variable Ordering* - The speaker has complete freedom in the order of presenting the description. This may result in information necessary to understand a statement being provided in the statements that follow it rather than in those that precede it.
- *Abbreviation* - An abbreviation is a term that stands in for a larger amount of information – larger in terms of either lexical size (i.e., the abbreviation is a synonym) or semantic content (i.e., the abbreviation is a cliché name).
- *Contradiction* - The input description is contradictory if a logical inconsistency can be deduced from it. Contradictions are indicative of problems in the input description.
- *Incompleteness* - The input description is incomplete when it is missing information necessary to fully describe the idea being acquired.
- *Inaccuracy* - The input description is inaccurate when it does not describe the idea the speaker wishes to convey.

The speaker uses these techniques for a number of purposes: to save time, to accommodate an imperfect understanding of what needs to be said, and to accommodate other limitations on the speaker's ability to provide a precise and correct description. Note that none of these informalities is due to properties of the input command language, which has a well-defined syntax and semantics (approximately first order logic). Informality is a property of the conversation as a whole and wholly attributable to the speaker.

Resolving informality is at the heart of what the Listener does. Informality is inevitable due to: the manner in which the speaker accesses the cliché library, limitations on the communication channel, and the basic formulation of the Listener's acquisition problem.

As described previously, clichés are used as the foundation of communication between the speaker and Listener. However, when the speaker refers to a cliché, he usually does not refer to the most specific appropriate cliché in that hierarchy of clichés, i.e., his statement is ambiguous. The Listener resolves ambiguities choosing appropriate, more specific clichés.

A restriction on the Listener is that the transfer of information occurs across a sequential channel. An immediate consequence of this sequentiality is that the speaker must impose an order on his utterances. Provision for variable ordering is made so that this decision has no impact on the final state of the acquisition. Communication is slow across this sequential channel, therefore the speaker uses abbreviation to shorten the description that needs to be presented. These interface limitations account for two more types of informality.

Finally, the Listener must always be in one of the following states during the acquisition. In this formulation of the problem, each of the three unsuccessful states of the acquisition corresponds to a type of informality.

- The corpus of facts deducible by the speaker is a subset of the corpus of facts deducible by the Listener, i.e., success.
- The corpus of facts deducible by the Listener contains a contradiction.
- The corpus of facts deducible by the Listener does not contain all of the facts deducible by the speaker, i.e., incompleteness.
- The corpus of facts deducible by the Listener is inconsistent with the set of facts deducible by the speaker, i.e., inaccuracy.

Informality hides information from the Listener. To fully understand what the speaker is saying, the Listener must recover this hidden information by resolving the informalities in the speaker's input description. This recovery requires a source of relevant semantic information to assist in the detection and correction of informalities. The cliché library is the source of this additional information.

The following subsections describe the Listener's approach to resolving each of the informalities listed above. (Note that the headings name the resolution technique rather than the informality, but the order follows the list above.) Each type of informality is handled separately. The suite of techniques is applied opportunistically and uniformally to the evolving description.

Word Disambiguation

Word disambiguation (or simply disambiguation) is the Listener's principal mechanism for fleshing out the description being acquired. In general, disambiguation refines the description making it more precise, leaving less room for (mis)interpretation. Statements made by the speaker often refer to high-level clichés (clichés towards the top of the hierarchy in the cliché library). As the description evolves, refinements of these higher-level clichés become applicable. Word disambiguation recognizes opportunities to refine a statement using the more specific cliché and incorporates the refined statement into the evolving description[2].

Each disambiguation made by the Listener increases the facts known to the Listener by the quantum of information in the cliché definition. For example, suppose the speaker says: "The *window* is broken," and then continues to elaborate on the problem. At some point, the Listener's disambiguation processing might decide that *window* referred to a *computer-window*, as opposed to *house-window*. This recognition causes the instantiation of the associated cliché resulting in a flurry of statements (e.g., this is a conversation about computers not houses, the object is an image on a screen, ...). This reduces the level of ambiguity in the Listener's corpus of statements and may provide the facts to trigger further disambiguations.

When disambiguating, the Listener has two options in deciding whether a cliché is applicable in a particular situation. It may be conservative and only make this determination when the target objects completely satisfy a set of preconditions required for an object to be an instance of the cliché, or the Listener may make a heuristic jump based on satisfaction of most but not all of the preconditions. As discussed in Section 3.2.3, the Listener makes use of both of these strategies, favoring the non-heuristic strategy. It makes use of the heuristic strategy when the word being refined would otherwise provide no additional constraint on the evolving description. In this case, disambiguation trades the possibility of making a wrong decision for the added knowledge due to making that decision. If this decision later turns out to be incorrect, the Listener can retract it and make a better choice.

Order Independence

Order independence permits the speaker to present input in whatever order seems

[2]For example, see command 4 on page 33 where *tracks* is disambiguated to *tracks-%tracking-system*.

most natural. Unlike human understanding, the ultimate understanding obtained by the Listener, is independent of the order of the speaker's exposition. (See Section 5.3 for an evaluation of order independence.) At any point in the conversation, information necessary for a more complete understanding of the description may be missing. The Listener will reach the appropriate conclusion once the final piece of information needed for a deduction is offered.

The Listener can ıetect certain obvious omissions of information, e.g., reference to an object before it is defined. In these cases, the Listener posts agenda items indicating its desire to be provided with this information (see figure 2-6 on page 34). The speaker can examine the agenda, at his leisure, and choose to address an agenda item to move the acquisition forward.

Support for Abbreviation

The Listener supports two forms of abbreviation. Use of synonyms, the simpler of the two, is purely syntactic. When defining an object, the speaker can declare a synonym for that object's name[3]. Synonyms are supported by input pre-processing and output post-processing. Since the abbreviation is purely syntactic, it does not effect the internal reasoning of the Listener.

The speaker may also use a general term in lieu of a more specific term to access a concept in the cliché library. This kind of abbreviation facilitates access to the cliché library since the speaker need not know the precise names of the many refinements of a concept. As described previously, word disambiguation serves to resolve the abbreviation to its referent once enough contextual information is provided.

Contradiction Handling

A contradiction occurs when a statement can be proved to be both true and false. Detecting conflicts assists in debugging the description. As the description evolves, each component object becomes increasingly constrained. The "shape of the hole" each object fits in becomes increasingly complicated. Bugs in the description can be detected when objects no longer fit into the holes to which they are assigned, thus causing a contradiction. When a contradiction arises, the Listener presents a set of error correction strategies that can assist in its resolution.

The simplest way to resolve a contradiction is to retract a premise underlying it. This, however, results in a net loss of information. For example, suppose there is a problem traceable to an underlying premise `X=6`. Retracting `X=6` eliminates the value of `X` and any deductions that follow from it. Alternatively, the contradiction can be fixed without a net loss of information, by replacing the premise with another one, perhaps with `X=7`. In this case, `X` gets a new value and any deductions that depended

[3]See commands 2 and 3 on page 32 for a use of synonyms.

on the old value are recomputed[4]. Depending on the syntax of the premise, there are different contradiction resolution strategies. Each strategy suggests a related premise to substitute for the problem premise. For example, if a premise asserts a type for an object, the type can be replaced with a sibling or parent type[5].

The resolution choices presented are a function of the current state of the description. However, the current choices may be incomplete with respect to a future state of the description. Therefore, permanently adopting a particular choice would violate order independence if the correct choice is currently missing. As the session proceeds, new viable resolution choices may become available. For each resolution strategy previously adopted, the Listener continues to monitor the changing description for additional resolution options. If any are found, they are brought to the speaker's attention[6]. Assuming that when eventually presented with the correct choice (for a previously selected resolution strategy), the speaker can identify the correct resolution strategy from among all those presented, order-independent understanding is maintained.

The Listener also supports the simple strategy of retracting a premise without replacement. This is useful when either no replacements are available or appropriate or when the speaker simply wishes to retract a statement.

The Listener enforces an expectation regarding the equivalence of terms with different names: distinct terms should not name the same object, i.e., there should be no accidental alignment of terms. The use of different names (except for abbreviation) should carry with it a corresponding difference in meaning and identity of the named objects. When two distinct terms name the same object, then there is a contradiction due to the misuse of a second redundant name[7]. This expectation is specific to the acquisition methodology implemented by the Listener and causes additional contradictions during description acquisition.

There are two ways to resolve an accidental alignment of terms. If the definitions should not be equal, then the normal contradiction resolution strategies can be used. However, if two names have been mistakenly used, then the reformulation mechanism (see description in Section 3.1.3) can be used to eliminate one of them, bringing the Listener to a state comparable to that it would have been in if the second name were

[4]Command 11 on page 41 shows the analyst examining replacement options for the records property of check-out.

[5]See command 19 on page 48 for an example of the type replacement heuristic.

[6]For example, in command 11 on page 41 the speaker is given choices for replacement values in an equality. Later, after command 15 on page 45, the Listener presents a new set of possible replacement values.

[7]Command 21 on page 51 shows the RA detecting two tracking operations, unshelf and unshelf-all, that have the same definition.

never used.

Augmenting the Description

The speaker's description is inevitably incomplete and the Listener tries to augment it. In general, incorporation of information from the cliché library augments existing descriptions. For example, word disambiguation augments the description with refined word meanings selected from the cliché library.

Incompleteness of object descriptions, i.e., missing role values, is easily detected and in some cases roles can be filled in by using the ***possible value heuristic***. The possible value heuristic searches for all instances of the type statically assigned to a role. This is the same heuristic used in contradiction handling to search for alternate values. In the case where a role is empty, the possible values are considered as alternatives to fill the role and the speaker may be asked, when the role in the cliché definition is annotated as being important to fill, to select one[8].

Missing role values may also be filled from context. The speaker may explicitly establish a context[9] that corresponds to a role of a particular object. Definitions made in that context may be augmented with information that states the defined object is to be included in the context role (if and only if the defined object is of an appropriate type for the context role)[10].

Incompleteness of a well-formed description, i.e., a description missing information orthogonal to the current state, is not handled. This is a validation (and elicitation) problem. Detecting incompleteness of this kind requires a source for global expectations regarding the evolving description. For example, in describing an automobile as a transportation-device, orthogonal information might include describing the auto as an environmentally-sound-machine. To detect this incompleteness, the source of global expectations would have to describe artifacts that are appropriately described as environmentally-sound-machines and provide some guidance as to when it was appropriate to view an artifact in this manner. The cliché library is a possible source for such information. However, providing such information is costly in terms of both the size and modifiability of the cliché library.

The cliché library contains locally defined (see Section 5.1), independent descriptions and it is not useful as a source of global expectations. Given the way the cliché library is structured, incompleteness processing would have to exhaustively search through and evaluate a large unstructured space of hypothetical worlds. The problem with structuring this space and providing a source of expectations is a tradeoff

[8]See command 29 on page 57 where a value for the monitor of a policy is chosen.

[9]See command 8 on page 36 where the speaker enters the functional-requirements context.

[10]See command 10 on page 39 where check-in is made a member of ULDB.functional-requirements due to the current context.

between creating general, modular, reusable clichés (that are easy to define) and having highly interrelated clichés (that are difficult to define) which permit exploring typical cliché patterns of interaction.

Checking Accuracy

The Listener detects many inaccuracies in the description in the form of contradictions. The cliché library provides an additional source of expectations that results in an increased number of contradictions. The contradiction handling mechanisms assist in correcting these inaccuracies.

However, the Listener only detects inaccuracies that manifest themselves as contradictions. No assistance is provided for the case when the description is well-formed but simply not what the speaker really means. In this case, validation is required to correct the problem. Validation of the description is supported by construction of a summary document, which can be reviewed for accuracy and consistency with the reader's expectations. In general, however, validation is not an automatable process [13] and is left to the speaker.

3.1.3 Support for Evolution

Support for evolution entails supporting changes in the description. As an acquisition session proceeds, the speaker may change his mind about any part of the description presented thus far. Such changes may be stimulated by feedback provided by the Listener, or the speaker may simply desire to change a previous input In any case, the idea must be allowed to evolve in whatever direction the speaker wishes to take it.

A description may evolve monotonically or non-monotonically. Incremental acquisition and incremental reasoning are the basis for both kinds of evolution. Incremental acquisition permits the speaker to explain an idea in stages rather than requiring an all-at-once explanation. Incremental reasoning ensures that only the representation of the changed (or added) components of the description are recomputed as opposed to recomputing the entire representation.

Non-monotonic evolution occurs when mistakes that have been made in the description are corrected[11]. The basic evolutionary capability to support non-monotonic evolution is the ability to retract a statement. The strategies which permit fixing contradictions in the description all rely on the ability to retract previous statements.

One additional capability provided to support non-monotonic change is a reformulation command. Reformulation allows the speaker to iterate through all input

[11]For example, in command 18 on page 46, the user changes the value of check-out.records from remove to borrow.

statements that use a particular term and selectively change occurrences of that term to another term[12]. The Listener retracts the selected old statements and asserts the new ones, incrementally updating its understanding to reflect the new statements

3.1.4 Incremental Feedback

The Listener provides the speaker with two kinds of feedback regarding the evolving description. First, the Listener paraphrases the speaker's input to indicate that it has understood the basic implications of the input statements. Second, the Listener announces interesting deductions to assist the speaker in understanding the implications of the evolving description and to trigger new insights in the speaker regarding the description being acquired. These reports bring utility to the Listener's deductions. The Listener could conclude the most intricate and significant facts, but if they were never reported they would be useless.

The difficulty in providing feedback is deciding what to tell the speaker without overwhelming him with data. The Listener could simply report each newly deduced fact, but there would be a great deal too many of them, most of which would be uninteresting. (In processing the example in Section 2.8, the Listener deduces over 10,000 facts.)

The Listener considers a certain syntactic subclass of deductions to be interesting. These deductions are further pruned by other criteria to reduce the amount of feedback. For example, the deduction of a new type for an object is considered interesting. When more than one new type is deduced, all but the most specific types are pruned from the feedback. Deduced types for roles are also reported. Role types are interesting when a specific value for the role has not been determined, since the types may constrain the set of objects that can fill the role. For example, `(Heavy (Weight X))` might imply a minimum weight of `X`.

The deduction of a slot value is also considered interesting. Since understanding a set of symbolic relationships can be difficult (see below), reporting a direct equivalence, e.g., `(= (weight flight007) 1000000)`, implied by the symbolic relationships below, is easier to understand and evaluate.

- (= (weight flight007) (+ (weight airplane) (weight passengers)
  (weight baggage)))
- (= (weight baggage) 40000)
- (= (weight passengers) (* number-passengers average-person-weight))
- ...

[12]Command 22 on page 51 demonstrates the reformulation of a type/token confusion between book (the type) and copy-of-book (the token).

Knowledge of the value is interesting for two reasons. First, it can be useful in evaluating properties of the object, e.g., is the plane too heavy to takeoff. Second, it reveals information about the state of the acquisition, i.e., enough information has been acquired to actually determine the weight of the airplane.

Note that when a role has a value, any report of the role type is pruned, since all the relevant information will be reported in association with the value.

Loss of information is also considered interesting because a goal of an acquisition session is to increase information content. If a role is reported to have a particular value and it loses that value without any replacement, the Listener will report the net loss of information[13]. The Listener also reports loss of types. The speaker may be surprised by such occurrences if he believes there is enough information to deduce the lost value. This observation may alert the speaker to a problem in the description.

Another interesting observation that the Listener makes is the use of a new term. When the Listener does not recognize a term, it prints it back inside of double quotes to indicate it has not been defined. The quotation continues until the speaker defines the term.

3.1.5 Summarization

The Listener can produce a document that summarizes the state of the description [34]. This document can be read and reviewed in a setting outside of the Listener system and therefore used as a medium for the transfer of information to other people concerned with the acquisition process.

The document has two important properties that increase its usefulness: precision and coherence. An automatically produced document is precise in a way that is difficult and tedious to achieve when writing a document by hand. Term usage, layout, and level of detail all follow the algorithm defined for document production. Thus, the reader can make reliable inferences based on detailed features of the actual presentation. This increases the information content of the document.

The document attempts to present the description in as coherent a manner as possible. Coherency is due partly to the removal of informalities in the input description and partly to a document organization independent of the order of data entry. The summary document also includes coherent descriptions of individual objects. These are produced using parameterized canned-text that is part of the additional information defined in a cliché.

Each chapter of the document is intended to be read as "flowing prose", achiev-

[13]For example, in command 18 on page 47 the value for check-in.records is lost. The speaker considers this odd because check-in is the inverse of check-out and therefore the inverse records value should have been propagated.

ing as much as possible a clear, linear presentation of the data. This poses a data presentation ordering problem for the Listener. However, since the description has been augmented with information from the cliché library and informalities have been resolved, everything that needs to be said in the document is known ahead of time. The document generator's job is to select and present the data in an organized manner. Clichés provide some organizing principles for the presentation of the data. For example, a cliché may include information about what order to describe roles in and which roles merit a separate document section for their description.

Each document chapter concludes with a dictionary of terms used in that section. The dictionary is provided both to acquaint the reader with unfamiliar technical terms that have precise meanings and to be a reference for someone analyzing the requirement. The use of a dictionary also permits the text to run uninterrupted by definitions of terms, improving the document's overall readability.

The document is only one view of the description. Tradeoffs are made in its presentation that would be made differently if it were, for example, a hypertext document. The current document is a linearized presentation of inherently interrelated concepts. The DKB contains justifications behind all facts that cause statements to be included in the document. Eventually, when the DKB is delivered along with the document, each statement in the document could be explained and analyzed in more detail.

3.2 Techniques

The capabilities described above are supported by the techniques explained below. The Listener's power derives from this ensemble of techniques working together producing the behaviors previously outlined. The description of techniques presented here captures the high-level design decisions made in constructing the Listener system, they include:

- CAKE - the reasoning system on top of which the Listener is built.
- cliché-frames - the representation of clichés.
- classification - a work horse for performing word disambiguation.
- plausibility checking - another mechanism supporting informality resolution.
- contradiction processing - the primary mechanism used to debug descriptions.
- the command language - the interface to the Listener.

3.2.1 CAKE - The Underlying Reasoning System

The Listener is built on top of the CAKE [35, 36] reasoning system. This choice of reasoning system determines two major system parameters: the basic underlying knowledge representation and the basic reasoning processes.

Facts are represented in a quantifier-free predicate calculus representation which is perhaps better thought of as a structured propositional calculus representation (i.e., propositions may refer to objects and relations). Structured propositions (from here on just propositions) have a truth assignment of either true, false, or unknown. These propositions are stored in a knowledge base called the DKB.

CAKE supports standard boolean logic semantics. Deduction is supported by a unit resolution algorithm that is not complete but produces, in linear time, basic deductions involving boolean connectives. A truth maintenance procedure ensures the propagation of correct truth values in the face of changing truth values.

CAKE provides several other reasoning procedures that extend and improve the coverage of unit resolution. Equality reasoning implements the semantics of a substitution notion of equality. Complete equality closure, using substitution of equals, is performed in polynomial time. Pattern-directed demon invocation is used to implement other kinds of reasoning. Demon invocation interacts with equality reasoning and completeness is assured in the sense that if performing an equality substitution would trigger a demon, then that substitution (but not all possible substitutions) is performed [37]. Demons are used as the basic technique for extending CAKE's reasoning abilities. For example, demon invocation is used extensively in noticing new and changing facts to be reported as feedback by the Listener.

In addition to supporting the basics of the knowledge representation and reasoning schemes, CAKE supports redefinition at the ground level of the reasoning system. Conclusions based on invalidated propositions are retracted and new ones are made with the appropriate modifications to the DKB being recorded.

CAKE also provides a base for incremental, order invariant deduction. The semantics of CAKE's logic are captured in a clause representation. A clause is a set of propositions and associated desired truth values that has the property that at least one of the propositions must have its desired truth value. A clause supports a deduction when all but one of its propositions have the wrong truth value thus determining the truth value of the remaining proposition. This clausal reasoning supports incremental changes in truth values and is order insensitive. The Listener builds on top of this and structures its reasoning so that these properties are preserved. Incomplete lines of reasoning are placed on an agenda and triggered by demons that react to the addition of necessary data. CAKE's truth maintenance mechanism assures that truth values are updated when all necessary data are available.

Since CAKE does not uniformly support quantification or conditional reasoning, it has been supplemented with predicates that permit the expression of such statements. These predicates include: !the-set-of-all, !there-exists, !for-all*, and !if. The Listener does not implement the full semantics of these predicates. Rather, it creates anonymous individuals to facilitate reasoning about the plausibility and meaning of a quantified or conditional statement.

For example, suppose the speaker states:
`(!For-All* (?N Number) (And (Evenp ?N) (Oddp ?N)))`. (`?N` names a quantified variable of type `Number`). The Listener creates an anonymous individual for `?N`, e.g., `N001`, and substitutes it for `?N` resulting in the following concrete statement: `(And (Number N001) (Evenp N001) (Oddp N001))`. The Listener proceeds to understand this statement which may result in: determining that `N001` is an integer (from the definition of evenp) and that the statement is self-contradictory. The use of anonymous individuals permits the Listener to remove the quantifiers from a statement and analyze the resulting concrete statement. The concrete statement is asserted with the quantified statement as its support and the Listener's understanding mechanisms are applied to it[14]. However, since anonymous individuals are meaningless to the speaker, any output containing them is pruned unless the output is notification regarding a contradiction.

In the case of a conditional statement, e.g., `(!if A then B else C)`, none of `A`,

[14]For example, in the report commands 13, 14, 15, and 16 (page 42), the contents of the reports are specified using quantified statements. In command 15, deductions from a quantified statement are propagated to the UL. In command 14, results are generalized to the type of the anonymous individual.

B. or C can be asserted to be true (assuming they start out unknown), but each is analyzed for plausibility since it must at least be possible for each of them to be true. Normally, only true statements are analyzed for plausibility, but in this case the analysis is extended in order to capture more of the meaning of the conditional statement.

3.2.2 Cliché-Frames

Clichés are represented using a frame construct called a *cliché-frame*. Cliché-frames represent three kinds of information:

- Definitional - Definitional information is asserted to be true of each instance of the cliché-frame.
- Computational - Computational information assists in performing some task. For example, *canned text* assists in the production of text to be included in a summary document.
- Predictive - Predictive information provides an index to the facts that will become true if an object is asserted to be an instance of the cliché-frame. It is used to help the Listener resolve informalities that require an abductive choice to be made.

Recall that a cliché consists of a set of roles and a set of constraints that relates the roles. Cliché-frames also have a set of roles and a set of constraints. An example of a cliché-frame definition is shown in figure 3-2. An explanation of the components of that definition follows.

```
(Def-Cliche Bicycle (Vehicle)
  (Def-Roles
    (Front-Gearing Gearing)
    (Rear-Gearing Gearing)
    (Number-of-Gears Integer))
  :Preconditions
   (Low-Gear ?Front-Gearing ?Rear-Gearing)
  :Consequents
   (< ?Maximum-Speed 60mph)
  :Default-Consequents
   (> ?Number-of-Gears 10))
```

Figure 3-2: The Bicycle Cliché-Frame

- Each cliché-frame has a *name* (here, in figure 3-2, **Bicycle**) which is the first argument in the **Def-Cliche** form. These names are symbols, which are used to provide access (as described later) to the cliché.
- *Parents* (e.g., **Vehicle**) of a cliché-frame are listed in the second argument. A cliché-frame inherits semantics from its parent cliché-frames as if the parent definition had been textually included in the child. For example, roles, such as **maximum-speed** in the consequents, are inherited.
- *Roles* are specified inside of a **Def-Roles** declaration either by name or by a name-type pair, e.g., **(Number-of-Gears Integer)**. Role values are referenced in constraints by a **?role-name** syntax.
- The *constraints* of the cliché-frame are divided into the following categories to help support different functionalities of the Listener. Constraints are statements that are true of every instance of the cliché-frame.
  - **:Preconditions** are used to support the classification algorithm (described in Section 3.2.3). Preconditions specify a set of constraints necessary for an object to an instance of a cliché-frame. The set of preconditions can be tagged as being complete in which case they define a set of sufficient constraints that can be used to deduce that an object is an instance of this cliché-frame. Implicitly included in the preconditions are constraints that state that the object is an instance of each of its parent types and that it has the appropriate set of typed roles.
  - **:Consequents** are asserted to be true of each instance of the cliché-frame. The syntax **?Self** refers to the object which is the cliché-frame instance. The syntax **?Foo.Bar** refers to the *bar* property of the *foo* property of the instance.
  - **:Default-Consequents** are identical to consequents except that they are supported by a default premise. When involved in the resolution of a contradiction, default premises are considered more likely candidates for retraction than others. The Listener can also use premise altering heuristics (described later) to substitute a different but related premise for a default.

Types of Clichés

The cliché library defines a number of ontological categories at its top-level, e.g., class, relation, and function. In a particular domain there may be additional ontological categories, e.g., in requirements analysis there are agents, actions, and systems. Any of these categories may be a parent of a cliché definition.

For the convenience of cliché definition (and implementation) certain syntactic sugar forms of definition have been provided for the different ontological categories. For example, consider the definition of a relation in figure 3-3.

```
(Def-Relation Borrower-Lending-Repository
              (?Object ?Lending-Repository ?Patron) ()
  :Defining-Terms
   (Lending-Repository ?Lending-Repository)
   (?Lending-Repository.Collection-Type ?Object))
```

Figure 3-3: The Borrower-Lending-Repository Cliché-Frame

This figure shows the definition of the borrower-lending-repository relation. The distinguishing feature of a relation is its argument list, which is defined in the second argument to the `Def-Relation` form. The argument list completely defines the structure of the relation cliché and no other roles can be defined. The arguments are, in effect, the roles and are referred to using the same syntactic convention as roles, e.g., `?Object`. Note that the argument list might have been defined as the value of an *argument-list* role. However, since it is central to the definition of the relation and since its static value is important in processing the cliché-frame definition (e.g., to interpret the constraints), the argument list is defined statically in the syntax of the cliché-frame definition.

A relation may have consequents and default-consequents. However, it uses `:Defining-Terms` as a replacement and extension of preconditions. Defining-terms are used as preconditions during classification. However, they are also used as an asserted constraint when an instance of the relation is known to be ***defined*** but not necessarily true yet. For example, if the speaker says:

(When (Borrower Mixer Cabinet Bob) (Making-Cookies Bob))

then the borrowing relation has not been asserted true, but it has been defined, i.e., it has been defined as a sensible possible occurrence. Therefore, the defining-terms on borrower are instantiated.

There are special forms for defining relations, functions, and actions. Each form provides syntactic sugar to help capture semantics particular to the kind of cliché. Nevertheless, each special syntactic form defines a full-fledged cliché.

More Cliché-Frame Semantics

In addition to the defining forms shown in figure 3-2, cliché-frame definitions may include the following additional defining forms. A uniform way to think about these additional defining forms is simply as additional constraints. Each form expands into definitional information (a constraint) and possibly into computational and/or predictive information.

- **:Fill-Roles (:Role-Name Value)** - declares that **role-name** should be filled with the designated **value**. This declaration can be used when a child determines the value of a role in its parent.
- **:Defaults (:Role-Name Value)** - is the same as fill-roles except that the value is made a default value. This is used to provide the standard notion of default role values.

The next form provides computational information that assists in contradiction detection.

- **:Equality-Views** - specifies sets of roles that define how to determine equality between two instances of a cliché-frame. This is used to help detect objects with different names and the same definitions.

The following forms are used to help in document production.

- **:Canned-Text** - Many kinds of canned-text can be defined on a cliché-frame. *Abstract* and *Overview* are the most common. Abstract is used when creating a high level summary of a cliché instance. Overview is used when creating a more detailed summary. The canned-text is a combination of words and references, for example.

  ```
  (!text ''*Self is an animal with *Prop1.Prop2 habits.'').
  ```

  ***Self** refers to the cliché-frame instance. ***Prop1.Prop2** refers to the **prop2** property of the **prop1** property of the instance. When a summary document is produced, it combines various kinds of canned-text depending on the required level of detail.
- **:Role-Sections** - lists roles in the cliché-frame that are to be described in their own document sections.
- **:Role-Order** - defines an order for presenting roles in the summary document.

Cliché-Frames for Requirements

In the requirements domain, cliché-frames are augmented with a set of constructors that form the basis of a preliminary *requirements-cliché calculus.* This representation provides a base for supporting the necessary requirements processing. The added constructors include a basic set of concepts fundamental to requirements descriptions [18]. See Section 2.5 page 23 for an explanation of the requirements ontology from which these constructors are derived.

Figure 3-4 shows the definition of the tracking-system cliché. A detailed explanation of the new keywords is provided to explain the information captured in this definition.

```
(Def-Cliche Tracking-System (System)
  (Def-Roles
    (Target Environment)
    (Target-State (Set-Of Defined))
    Item-States
    (Manner-Of-Observation Manner-Of-Observation)
    (Data-Observed Data-Observed))
  :Actions
   (Def-Action Track-Target)
  :Behaviors
   (Def-Behavior Track-State)
   (Def-Behavior Request-Change)
  :Invocation-Protocol
   (Execute ?Self Track-State)
   (Execute ?Target.Interacting-Agent Request-Change)
  :Default-Consequents
   (= ?Item-States ?Target.Item-States)
  :Consequents
   (Tracks-Tracking-System ?Self ?Target)
   (= ?Target-State ?Target.Item-States)
  :Product-Functions
   (Track-Target-State ?Self)
  :Canned-Text
  :Overview
   (!Text ''The main purpose of *self is to track the state of physical
    entities. *target.$agents act in the environment causing changes to
    the item-states of the *target which are reflected in *self.''))
```

Figure 3-4: The Tracking-System Cliché-Frame

- **:Actions** - Actions that a system can execute are named inside the cliché-frame body. They are defined in detail outside of the frame with a **def-action** form (the definition amounts to a definition of the action's pre and post-conditions, input and output expectations, and other properties related to the operational nature of the action).

- `:Behaviors` - Behaviors that a system can execute are named inside the cliché-frame body. The current RA version does not support defining their semantics in detail. This designation is used primarily in concert with the `:Invocation-Protocols` to guide the refinement of objects that execute certain behaviors.
- `:Invocation-Protocols` - An invocation-protocol defines which *agents* typically execute which behaviors. This information provides a predictive index into the cliché-frame.
- `:Product-Functions` - A product-function defines a capability relevant to the system that needs to be supported by some component of the system.

Other constructors that can be used in a requirements cliché-frame definition are:

- `:Agents` - Agents associated with a system are named in the cliché-frame body.
- `:Functional-Requirements` - A functional-requirement describes a solution provided by a particular cliché-frame to problems defined by the product-functions of another cliché-frame.

Operational Properties of Cliché-Frames

Each time an object is determined to be an instance of a cliché (whether asserted directly by the speaker or as the result of an informality resolution process), the cliché-frame constraints are propagated to that object. This results in a flurry of assertions. These assertions are processed as if they had been stated by the speaker. In particular, the mechanisms for informality resolution are uniformly applied to all assertions regardless of their source. Because of this, the cliché library is not forced to compile out all possible interactions of cliché-frames. Thus, the cliché library can encode informalities to be resolved at runtime. This makes construction of the cliché library a more modular process (see Section 5.1) by avoiding the need to define a cliché in all possible relevant contexts.

The Cliché Library

The hierarchical organization of clichés in the cliché library generally reflects a library construction methodology that organizes cliché specializations around some *axis of refinement*. Certain groupings of children clichés reflect choices of a parameter setting along the axis of refinement. The speaker explicitly or implicitly traverses these refinements as the description becomes better developed. For example, figure 3-5 outlines a fragment of a cliché library. A *comparison-system* compares two objects according to some predicate. Its specializations: monitoring, tracking, and control, extend the comparison function with varying degrees of influence on the objects being compared. The degree of influence is, in this case, the axis of refinement. A *monitoring-system* reports on differences between the two objects (e.g., an airplane's variance form its flight plan). The next two clichés distinguish one object as being the real object, i.e., it has an independent physical existence, and the other as a virtual object, e.g., a memory image of the state of an object. A *tracking-system* alters the state of the virtual object to record an image of the real object. A *control-system* physically alters the state of the real object to conform to the state of the virtual object.

```
                    Comparison-System
                   /        |         \
                  /         |          \
                 /          |           \
                /           |            \
Monitoring-System    Tracking-System     Control-System
```

Figure 3-5: Related System Clichés

3.2.3 Classification

Each cliché has a different connotation from its parent clichés. This distinction is captured in its definition. The cliché definition consists, in part, of preconditions that provide applicability criteria of the concept to a particular context. When the speaker says "Foo add(ed) Bar to Baz," the Listener applies its current knowledge of Foo, Bar, and Baz along with the cliché library entries under *add* in an attempt to more fully understand the statement. For example, it may be the case that Foo is an agent that added the correctly typed (relative to the collection Baz) object Bar to the discrete-collection Baz in which case *add-object-to-collection* is an appropriate sense of 'add' since the preconditions of *add-object-to-collection* are all satisfied. Or, it may be the case that there is not currently enough information to justify the classification of 'add' to a particular cliché specialization. In this case, the Listener holds off on making such a deduction until more information becomes available.

Classification is an important mechanism used to incorporate knowledge from the cliché library in appropriate circumstances. It supports the previously described word disambiguation capability. Classification is invoked in three circumstances in the

Listener system: by word sense resolution, by plausibility checking (described in the next numbered section), and independently as an informality resolution mechanism.

Word Sense Resolution

A potentially important representation issue is how words in the speaker's statements refer to clichés. A simplification has been made in which the spelling of the words used in the input statement indexes into the cliché library. Thus it is important that the speaker says "X add(ed) Y to ..." versus either "X augmented Y by ..." or a similar statement in Spanish. A lexicon-based approach to concept indexing could be added to the Listener. This would, for the most part, separate the definition of a cliché from access to it. The lexicon would manage the initial match of words in the statement to high-level concepts in the cliché library. Currently, however, it is assumed that words in the input statements name concepts in the cliché library.

In understanding the statement "*add* x to y," the Listener attempts to deduce an appropriate sense for *add*, choosing from among the terms in figure 3-1 (*add-numbers*, *add-complex-numbers* ...)[15].

In order to perform word sense resolution, candidate ambiguous words must first be identified. For example, in "*add* x to y," the Listener only tries to disambiguate *add* and not either of the words 'x' or 'y'. Ambiguous words are designated in the cliché library and are known as *root terms*. The set of root terms consists of the tops of each tree of clichés.

Root terms provide a lexical index into a set of clichés related by some generic definition. The root term itself may have little or no definition. If the speaker can access the root term by name, this index allows the speaker to access the subordinate clichés without needing to know their precise names. The Listener will disambiguate the sense of the term[16], selecting the best candidate, as determined by the classification algorithm (heuristic version), from among the choices.

The Classification Algorithm

Classification provides access to domain knowledge, which is used to further the Listener's understanding of the description. In the Listener system, classification is a process by which statements are refined, using the cliché library, to imply a more specific form of the statement. The more specific the classification the better the understanding since a more detailed cliché definition can be propagated to it.

The Listener formulates the classification problem in the following sequence of

[15]Or, for example, in command 15 on page 45, the sense of ***borrower*** is resolved to ***borrower-lending-repository***.

[16]For example, see command 4 on page 33 where ***tracks*** is disambiguated to ***tracks-%tracking-system***. In this case, it may be reasonable to expect the speaker to know the term ***tracks***, but he is unlikely to know the term ***tracks-%tracking-system***.

steps.

- generate candidates - Each statement generates a candidate set of classifications. This set consists of all statements obtained by replacing the operator (in propositional form, the operator is the first term in the proposition) of the original statement with one of its immediate specializations in the cliché library.

  There are two kinds of statements involved in this generation: boolean propositions and non-boolean expressions. When the statement is a boolean proposition, e.g., **`(Spouse Mary John)`**, the refinement is also a boolean proposition, e.g., **`(Wife Mary John)`**. The result of a successful classification would be **`(Wife Mary John)`** being true and supported by **`(Spouse Mary John)`** and other appropriate preconditions. When the statement is a non-boolean expression, e.g., `(Plus X Y)`, the refinement is also a non-boolean expression, e.g., `(Integer-Plus X Y)`. The result of the classification would be `(Integer-Plus X Y)` being *equal* to `(Plus X Y)` and this equality supported by the appropriate preconditions. Having distinguished between the boolean and non-boolean case, the rest of the discussion will describe classification as if there were just a boolean case. The corresponding details of the non-boolean case should be clear.

- prune candidates - Each cliché defines a set of preconditions that is used to form a metric of the cliché's applicability. The set of candidates is pruned by eliminating those that have a precondition that is currently false. A candidate is also removed if it is currently true (the classification is done) or false (the classification would cause a contradiction).

- rate and select candidates - The remaining candidates are rated by counting the number of satisfied preconditions and the number of total preconditions. Candidates are chosen for assertion based on the percentage of satisfied preconditions. Depending on the process that initiated the classification, different selection algorithms are used. These are described below.

There are two similar kinds of classification that the Listener performs: definitional and heuristic. When a cliché has a complete sufficient definition and that definition is satisfied 100% then *definitional classification* is performed. The object is classified based on the following kind of deduction:

If (X is a T) and Precondition-1 and Precondition-2, then (X is a T1).

The second form of classification is *heuristic classification*. If a cliché only has necessary conditions, then there is no definition to be completely satisfied. (By default, the set of preconditions on a cliché is assumed to be incomplete, i.e., an anonymous, unsatisfiable precondition is included. The set can be designated complete by including a special precondition, `:no-more`, in the enumeration of preconditions.) In this case, and in cases where a definition exists but is not completely satisfied, the rating is less than 100%. Depending on the process that generated the initial candidate set, a heuristic choice may be made that chooses the single best candidate from the ranked alternatives. This comparison is based on both a comparison of the percentage of satisfied conditions and a comparison of the total number of conditions satisfied. In this case, support for the classification depends on data outside of the cliché definition and is captured in a premise that represents the Listener's heuristic guess:

If (X is a T1) and Precondition-3 and Heuristic-Guess, then (X is a T2).

In general, the Listener is disposed to performing mainly definitional classification. Heuristic classification is limited to word sense resolution. Provision has also been made for use of heuristic classification in an interactive mode of Listener operation. Currently, however, this mode is disabled to conform with the design goal of unintrusiveness (see Section 7 for a discussion of the merits of a more interactive Listener interface).

Word sense resolution employs heuristic classification since root terms convey little or no information until specialized further. The risks of a mistaken classification are tempered by the support for evolutionary description and conflict repair.

Classification of a statement does not halt once the statement has been classified to a level one deeper in the cliché hierarchy, since there may be additional applicable clichés. Further, the set of specializations of a cliché is not, by default, a partition of the class. The specializations of a class may include multiple partitions along different axes, e.g., for integer, a set of specializations might include: positive-integer and negative-integer (disjoint), and even-integer and odd-integer (disjoint). The specialization also might include *mix-in* properties of the class, e.g., for integer, possible mix-in properties include prime-integer and perfect-integer. Disjoint class relationships are used to prune out possibilities, e.g., once a number is determined to be odd it cannot be even. But an even number may still be positive or perfect. Given this kind of specialization configuration, multiple refinements of a proposition are possible.

Classification performs deduction that expands the description being presented by the speaker. It is a kind of recognition algorithm that opportunistically applies information in the cliché libraries to the current description[17]. Classification causes all statements to be associated with the most specific relevant cliché implied by the

[17]Command 9 on page 37 shows the ULDB being classified as a *%tracking-information-system* after having first been classified as a *%tracking-system* (in command 4) and as an *%information-system* (pruned from the feedback in command 9 by the paraphraser to reduce redundancy).

evolving description. For example, suppose someone is describing a vehicle used for daily transportation and that the Listener has deduced that it is a bicycle. The speaker then states that the vehicle's third front chain-ring has 28 teeth. This might cause the Listener to classify the vehicle as a more specific subclass of bicycle, touring-bicycle, since only touring bicycles have such a low front gear.

3.2.4 Plausibility Checking

Clichés are indexed by a number of predictive relations that are used to determine the applicability of a cliché to a certain situation. For example, given a role name, the disambiguation algorithm can obtain a set of candidate clichés whose instantiation would make the role defined on the object in question. In general, these relations are used when the Listener wishes to perform abductive reasoning to explain a particular proposition (e.g., to resolve an informality). To do this, the Listener needs to know which clichés, when applied to an object, will result in the proposition being true.

In the requirements domain, an invocation-protocol is associated with a system cliché and describes which agents execute which behaviors. When the RA observes that a particular behavior is executed by an agent, it must understand this statement as a plausible assertion. In this case, plausibility is defined as being the kind of agent, as defined by the invocation protocols, that can execute the behavior[18].

Each statement and its sub-expressions must be plausible – they must *make sense*. For example, given `(room-assignment (= (room jeff) 518) high-priority)`, it must make sense for jeff to have a room (perhaps Jeff is a student in a dorm), for `(room jeff)` to be equal to 518, for room-assignment to be a relation on the given arguments. and for each atomic expression to be a known object. Plausibility checking fills in background knowledge that is necessary to understand a statement.

Plausibility checking can be divided into two classes of processing: type inference and abductive reasoning. A simple example that captures the flavor of plausibility checking is as follows: if "the height of X is 68 inches," then type inference requires X to be the kind of thing for which it makes sense to talk about X's height, e.g., X is a physical-object. Abductive reasoning might further infer that X is a human, if humans typically have heights in the appropriate range.

Type Inference

Type inference assigns a class to an object based on the object's participation in certain relations (currently only property relations). For example, whenever a statement refers to a property of an object, the object must be the kind of thing

[18]The deduction in command 29 on page 57, "ULDB is an Advisory-System," results from an implication from the command statement that the ULDB must execute an authorization function.

that can have that property[19]. Information to support type inference is derived from definitions in the cliché library. For example, the definition of roles in the cliché library creates an index that lists which roles are defined on which clichés. This index is used to form a set of candidate clichés for an object that has a particular property referenced. The object is then classified as an instance of one of these clichés to make the role make sense and to make the expression be properly typed. (Type inference serves as a kind of word sense resolution for words that name properties.)

Abductive Reasoning

Abductive reasoning is a form of heuristic reasoning that asserts *causes* based on the observation of *symptoms*. The main symptom that can be exhibited by a description is that a statement *makes sense*. In fact, any statement made by the speaker is implicitly taken to make sense. Each kind of expression may have a different procedure associated with it that can find causes for a statement to make sense. These procedures look for (and may assert) statements that will imply the sensibility of the statement under consideration, i.e., they will try to cause the statement to be sensible.

Many kinds of statements make sense simply by their assertion, e.g., declaring that an object is in a particular class or stating that two things are equal requires no additional information. Some of the information required for a statement to make sense may be captured in the definition of that kind of statement. For example, for `(+ A B)` to make sense, `A` and `B` have to be numbers. The fact that `A` and `B` are numbers is included in the definition of `+`. This is a simple deduction, no abduction is required. However. other information may be required in order to effectively make sense of a statement. For example, in order for `(+ A B)` to be computed, `A` and `B` must be of compatible types, i.e.. one of the things that may cause the sensibility of `(+ A B)` is `(coercible (type-of a) (type-of b))` (another would be **`(same-type A B)`**). The plausibility check defined on `+` might search for a way to make sure that `A` and `B` are of compatible types and instantiate a cliché to make it so. This checking might, for example. supplement `(+ 3 4.1)` with the information that `(= 3 3.0)` permitting real number addition to be performed.

The RA includes the following plausibility checks. Whenever it is asserted that **`(Executes <Agent> <Behavior>)`**, it must be the case that the **`<agent>`** be capable of executing that **`<behavior>`**. The invocation-protocols provide an index that can be used to determine a classification of the agent that will imply the capability to execute the behavior.

Another example that the RA handles is deducing a solution (functional-requirement) for a particular problem (product-function). When the RA discovers a product-

[19]Part of the reason that so much is deduced in command 9 (page 37) is that the existence of the 'records' property allows the RA to classify check-out as one of the subclasses of tracking-operation.

function that is part of the evolving requirement, it looks for candidate solutions to that problem. Again, the cliché library sets up an index from problems to possible solutions. The RA finds a set of candidate solutions, tries to eliminate all but one of the candidates, and, if successful, chooses the remaining one to solve the problem.

Plausibility checking facilitates abbreviated communication by helping to fill in details that are required to reason formally about a statement but are considered implicit in its statement. However, the Listener makes limited use of these mechanisms because much of the reasoning that needs to be implemented this way is motivated by application-specific analysis procedures. For example, each requirements analysis has the potential to introduce a new set of analyses necessary to the domain of the requirement (e.g., finite state model analysis or accuracy evaluation). A more complete study of requirements analysis will be required to identify a usefully complete set of analysis procedures. The Listener concentrates on expanding the description in general, not on problem specific analysis. Nevertheless, the hooks for this type of reasoning are useful as a way to define new informality resolution procedures.

3.2.5 Contradiction Processing

The Listener captures all the contradictions that occur while processing an input statement. If any contradictions are found, the Listener takes the following steps (after the input statement has been completely processed):

- characterizes each contradiction by performing a simple analysis of it
- selects a contradiction to work on first
- explains the selected contradiction
- presents assistance for correcting the selected contradiction
- prunes the list of outstanding contradictions and, if any remain, offers to continue the above process selecting a new contradiction to work on

Characterization and Selection

The analysis of a contradiction involves obtaining the set of premises underlying the contradiction and separating out the *uninteresting* premises. The source of a premise determines whether or not it is interesting. In particular, "bookkeeping" premises from the Listener's internal reasoning are not considered interesting. Also, since the cliché definitions are assumed to be correct, i.e., any mistake is presumed to be a mistake in cliché selection not definition, premises from the definitions of clichés are uninteresting.

The interesting premises are rated as to how many of the current contradictions they participate in. Those premises that participate in the most contradictions are good initial candidates for retraction.

Working under the assumptions that understanding a contradiction is the most difficult part of the overall process and that a smaller set of underlying premises corresponds to a smaller and simpler explanation, the Listener chooses the contradiction with the smallest number of interesting underlying premises to work on first.

Explanation

The full explanation of a contradiction consists of an explanation of the truth value of each node involved in the contradiction. This consists of a forest of detailed dependency trees which may be difficult to understand because they reflect both the implementation details of CAKE and the convolutions of the Listener's own reasoning processes. However, each dependency tree also has a set of underlying interesting premises at its leaves that support the node. This group of premises provides a simpler, albeit less detailed, explanation of the node.

The explanation of a contradiction provided by the Listener consists of an enumeration of the nodes that participate in the contradiction, a short textual documentation of why these nodes are in contradiction, and a list of the underlying interesting premises. The speaker tries to mentally reconstruct an explanation that provides the information needed to fix the problem by working top-down from the participating nodes and bottom-up from the underlying premises. This reconstruction requires the speaker to focus on and identify the heart of the contradiction without being distracted by the convoluted dependency trees and thus assists the speaker in obtaining the information needed to correct the problem. (Of course, the dependency trees are available for browsing if desired.)

A topic that needs further exploration is how a dependency tree can be presented to get at the heart of an explanation. A certain amount of syntactic pruning can be performed on the tree, e.g., one does not need the explanation that `(And A B C)` is true because `A` is true and `B` is true and `C` is true. More important is the notion that each part of the tree and each premise has a particular commitment associated with it. For example, they may support varying numbers of other propositions. Also, certain facts are more changeable than others. Automating this sort of analysis would be a challenging project that would make the dependency trees much more accessible to the speaker.

Correction

Contradictions can be corrected by retracting a supporting premise. (See [38] for a more complete algorithm that recognizes that a contradiction may have multiple support paths and therefore a spanning premise set may have to be retracted.)

This retraction results in a loss of information. To counteract this tendency the Listener provides information preserving correction strategies [39] for different kinds of premises. There are four strategies, each accesses the cliché library and provides domain-guided assistance in resolving contradictions.

- Certain propositions express information in which specific values in the proposition are tentative and could be replaced by other related values. For example, a statement that declares the value of a parameter, e.g., `(= stack-size 1000)`, conveys two pieces of information: stack-size has a value and that value happens to be 1000. If this premise supports a contradiction, then to minimize information loss, it is useful to maintain a value for the parameter but not the specific value 1000. When a source of information to identify other candidate values is available, the Listener searches for such values and presents the speaker with a choice of replacement values that can be used to correct the contradiction. For example, if the premise is an equivalence that can be understood as `(= <role-of-object> <value>)`, then an alternative value can be selected from the instances of the type declared for the role in question[20].

- If the premise is a class assertion, a replacement class is searched for. The replacement class can be a sibling class (one that has the same parent) or a parent class. A sibling class is potentially as detailed (i.e., contains as much information) as the current class, whereas the parent class is likely to relax some constraints[21].

- If the premise is a *choice* (the Listener makes many kinds of choices: user choices, disambiguation choices, alternative value or class choices — all are represented by choice premises) then the choice can be retracted. Since the choice led to a contradiction, it will become false. The processing that originally caused the choice to be made is reactivated and another choice is made. This feature helps correct for any over aggressive choices the Listener makes in classification or resolving informalities[22].

- If the premise is a default then the correction strategies can be applied recursively to the proposition supported by the default. Normally, default values might be silently retracted. The Listener requires user intervention, however,

[20]Command 18 on page 46 shows alternative slot values being selected.

[21]Command 19 on page 48 shows alternative classes being selected. The first choices are sibling classes, the last is the parent class.

[22]Command 28 on page 56 shows the retraction of a choice and the deductions from an alternative option being chosen.

since a replacement option should be selected, if possible, to minimize information loss[23].

Contradiction detection is the Listener's primary bug detection mechanism. The user is expected to discern the underlying reason for the contradiction (little assistance is provided for this task except for the presentation of data) and come up with an appropriate repair strategy (with assistance from the Listener). Once a strategy is chosen, the Listener executes it and incrementally updates all affected decisions, reporting the results of the change and whether or not the contradiction has been fixed.

3.2.6 The Command Language

The command language determines how the speaker talks to the Listener. To avoid dealing with natural language issues, the command language has a precise, unambiguous syntax. The command language permits informality in that it allows fragmentary, semantically ambiguous and underspecified statements to be made. The set of commands is described below using a BNF syntax.

(Find-Requirement <Requirement-Name>) - informs the Listener that the speaker is starting (or resuming) a requirements analysis[24]. This command is only used once per session.

```
(Define <Name> {<Modifier>})
<Name> ::= <An-Atom> | <An-Atom>.<Name>
<Modifier> ::= <Type-Name> | :Synonym <Synonym> | :AKO <Type-Name> |
               :Member-Roles <Role-List> | :Defaults <Role-Value-List> |
               :Roles | <Role-Value-List> | :Card <Cardinality> |
               :Unique-Id <Boolean>
```

Define permits the definition of objects, classes, and roles. In an object definition, class information and role values may be included[25]. In a class definition, parent classes and role specifications may be included. In a role definition, class and arity information may be included. An abbreviation for the defined object may also be included in any of definition.

[23]Command 19 page 48 shows the retraction of a default and the recursive application of the class altering heuristic.

[24]See command 1 on page 31 for a more detailed explanation.

[25]See command 2 on page 32 for a more detailed explanation.

`(Fill-Role <Dotted-Role-Name> With <Value>)` - permits the speaker to assert an equivalence between the role of a cliché instance and some value or expression.

`(Go-To-Context <Context>)` - pushes into a new context[26].

`(Reformulate <Old> As <New>)` - iterates through every previous command using the first argument, offering to replace each occurrence of the first argument with the substitute second argument, and then reexecutes changed commands[27].

`(Resolve-Conflict <Number>)` - runs the conflict resolution procedures on a previously deferred conflict.

`<A-Logical-Assertion>` - commands with unrecognized names, i.e., not a member of the set enumerated above, are taken to be logical assertions. Arbitrary assertions can be stated in this manner.

[26]See command 8 on page 36 for a more detailed explanation.
[27]See command 22 on page 51 for a more detailed description.

3.3 Implementation

This section describes the architecture and implementation of the Listener system. The goal of this section is to supplement the techniques description with enough information to enable the reader to implement this Listener system.

Figure 3-6 shows an outline of the design of the Listener system. Rectangles denote software processes and diamonds denote data sources. The components of the system are described individually in the following sections. An overview of the entire figure is provided below, with references to the figure displayed in italics.

The top of the figure shows the two input data sources for the Listener. Before an acquisition session, the Listener is loaded with a *cliché library*. The cliché library is *compiled* into a form suitable for processing by the Listener system. The *library interface* manages access to the compiled library and supports instantiation of clichés during an acquisition session. The input for a session consists of a sequence of *commands* that are interpreted by the *command interface*.

The internals of the Listener system consist of the *informality resolution* component, the *paraphraser*, and the *contradiction handler*. These mechanisms act in concert and respond to the evolving state of the DKB. Informality resolution reads and writes data to the DKB. The paraphraser only reads data. Contradiction handling primarily reads data thought it may write data to resolve a contradiction. These mechanisms are built on top of the CAKE reasoning system. An *interface* is provided that permits a more natural (for this application) interaction with CAKE and permits monitoring this interaction. CAKE itself is augmented with a number of reasoning techniques to support the capabilities of the Listener.

CAKE's knowledge-base comprises the DKB. The *document generator* extracts the information it needs to generate the summary document from the DKB and the cliché library.

The Listener is implemented in Symbolics Common Lisp on a Symbolics Lisp Machine. It is implemented on top of CAKE which runs on the same platform. An introduction to CAKE terminology is provided to facilitate understanding of the discussion that follows.

CAKE Primer

A *term* is the basic unit of data in CAKE. Terms that represent propositions have a *node* associated with them that stores the truth value of the proposition. A node's truth value may depend on other nodes, these form the *support* of the node. A node that is true (or false) with no support is called a *premise*. A node is made true by *asserting* it (false by denying it). *Retracting* a premise makes it unknown.

Some terms are *values* (also called literals). Values have the property that no two values may be equal (e.g., the term representation of the numbers 1 and 2 are not

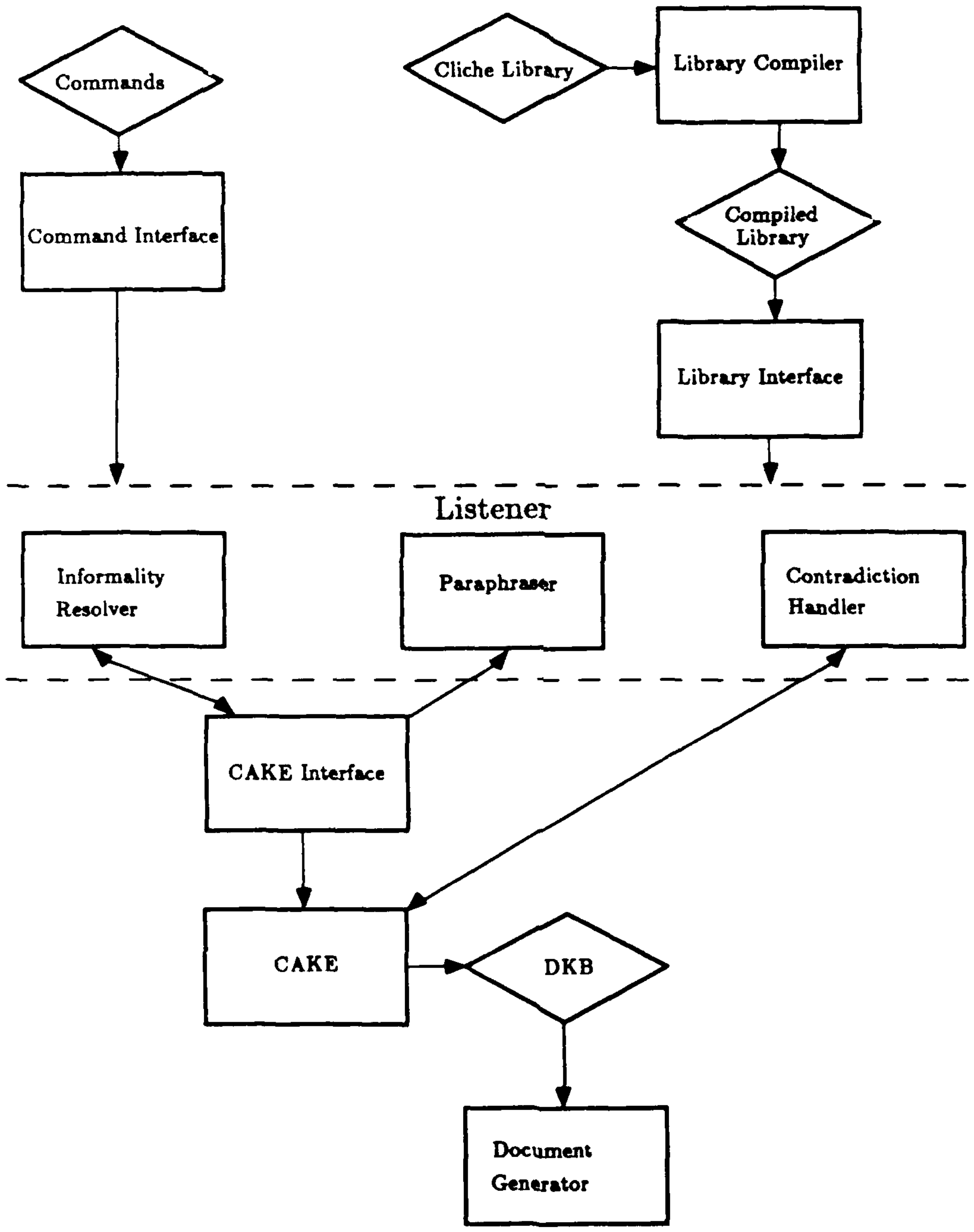

Figure 3-6: Listener System Software Architecture

equal). Many terms correspond to Lisp lists (others correspond to numbers, strings, or symbols). In list terms, substitution of equals is valid on all subterms, e.g., if `(= a b)`, then `(f b)` equals `(f a)`. Terms whose first subterm (the operator) is a *prefix* (a symbol beginning with a '!') are not transparent to substitution of equality.

CAKE uses a typed logic. A number of built-in types are defined, e.g., integer and number. The types boolean and defined are especially important. All terms in the Listener should be of type defined. Certain reasoning techniques will not reason correctly about a term unless it is defined. All terms that are asserted as propositions must be of type ***boolean***. Boolean constraint propagation will not work with terms that are not boolean.

Reasoning in CAKE can be extended through the use of pattern-directed invocation which is implemented by *noticers*. One kind of noticer is comprised of a triggering pattern and a noticer function. These noticers are run whenever a term is created that matches the triggering pattern for that noticer. Another kind of noticer responds to the change in truth value of a node. To maintain the correct interaction of noticers with the equality mechanism, noticer functions must be *transparent*. A transparent noticer has the same cumulative effect on the DKB regardless of which member of an equality class it is invoked on (as long as the member matches the triggering pattern). In other words, all dependencies on the syntax of a term must reside in the pattern and not in the noticer function [37].

An *overlay* is a function from one type to another. The application of an overlay to an instance of the domain type is an instance of the range type. The overlay also sets up correspondences between properties of the two types.

Finally, one word regarding the efficiency of CAKE is appropriate. To a first approximation, the most expensive act in the reasoning system is creating a term (due to interaction of noticers with term creation). The more terms there are in the system, the slower it runs. Therefore, creating terms indiscriminately is inadvisable.

(The footnotes in this section provide pointers to the files that actually contain the described functions. Most readers can completely ignore them.)

3.3.1 Command Interface

Input to the Listener is presented to the command interface[28]. Commands are parsed and translated into the appropriate CAKE interactions. Command processing consists of:

- expanding synonyms
- creating the premise support for a command

[28]See files: Commands, Rkbcoms, Top, Env, and Lib.

- recording the current context
- dispatching to and executing the correct function
- performing the clean-up and reporting functions

The first processing step on a command is expanding each occurrence of a synonym. The rest of the processing never sees the synonyms. Next, in preparation for translating the command, a *supporting premise* is created and asserted. This premise is used as support for any assertions made as part of the translation of the command. Structuring the dependencies in this manner permits the entire command to be retracted by retracting its supporting premise. Next, the context the command is being executed in is permanently associated with the command (commands are numbered and stored). This permits any reexecutions of the command to be made in the original context. Finally, based on the first word in the command, the appropriate command routine is dispatched to.

Each command routine is responsible for: parsing the command and extracting its arguments, creating any new cliché instances (according to the semantics of the command), and asserting propositions (with the support of the command supporting premise) that capture the intended meaning of the command. After the command is executed and its assertions are processed through the reasoning system, a *clean-up* routine is run to execute the rest of the Listener's reasoning processes. Finally, as part of the clean-up, the paraphraser reports its observations to the speaker.

One of the reasons that clean-up and reporting are separated from the normal processing of the reasoning system is to provide a way of executing functions during a quiescent state of the knowledge-base. This is necessary, for example, since truth values of propositions may change many times during the propagation of boolean constraints before settling on a final value. Reporting intermediate values to the speaker could cause confusion. The clean-up and reporting process is a multi-stage process which consists of:

- running epilogue procedures and checking wishes
- checking expression types
- checking symbolic assertions
- performing classification and plausibility checking
- checking for context insertions
- making observations and printing appropriate ones
- checking for new choice information
- checking for and resolving conflicts

The clean-up process executes the above steps, in order, repeatedly until none of them make any changes to the state of the knowledge-base. (Actually, all the steps before printing observations are run repeatedly until none of them do anything, then the observations are made, and then the remainder of the process is run again looping to the beginning if there are any changes to the knowledge-base. This provides the speaker with a description of the effects of a command before asking him to resolve any conflicts that may have arisen.) This ensures that each command has its maximal effect propagated and reported before executing the next command. Some of the clean-up steps will be described in more detail later. A brief description of each is provided here.

The epilogue holds function calls that need to be run in a quiescent system state. Generally, these functions interact with representations outside the reasoning system and thus, if for no other reason than efficiency, need to be executed when stable truth values are in place. Wishes are a technique used for informality resolution. A wish represents a desire for a particular node to have a particular truth value. For nodes that do not have the correct truth value a process is run that attempts to cause the node to have the correct truth value.

Checking of expression types is another informality resolution technique (quite specific to the implementation on top of CAKE) that is used to ensure that an object or expression is an instance of the appropriate (as deduced by syntactic checks) type. The expression is asserted to be of the correct type if it is not currently. For example, all assertions must be of type boolean. This process helps recover information implicit in the speaker's commands.

For efficiency, cliché definitions are not instantiated on all terms and generally only instantiated on terms that represent object names (which are also generally values). During reasoning, the system creates many symbolic references, e.g., `(pilot (first-airplane (airline-owned-by trump)))`, that may be deduced to be an instance of some cliché. If the cliché definition is instantiated on these symbolic instances, new and larger symbolic references may be created. Instantiating cliché definitions on these instances may create even larger symbolic references. The process may even trigger an infinite loop when (mutually) recursive cliché definitions exist, e.g., `(inverse (inverse (inverse ...)))`. Generally, the system waits until it discovers an object name as the referent of the symbolic reference and then instantiates the cliché definition on the named object. The equality system ensures that all appropriate deductions are made once the data is asserted on one member of the equality class. This technique may cause the Listener to miss some deductions (though in practice this has not happened), but it has proved necessary to control reasoning in CAKE.

Classification and plausibility checking are built on top of CAKE and have to be invoked at appropriate times. The bookkeeping regarding which expressions are candidates for classification and plausibility checking is stored as propositions in the knowledge-base. The classifier uses this information and the structure of the cliché library to check for new classifications. Any new classifications it makes are installed with the appropriate dependencies. Similarly, plausibility checking is triggered from these bookkeeping expressions.

The Listener implements a very simply context mechanism which results in the object currently being defined being inserted into the current current context role if they are of compatible types. This needs to run as a separate check (after the object has become an instance of all the appropriate types).

Various pieces of the code make observations and request that these observations be displayed to the speaker. During clean-up, the observations are checked to see if they still apply, are pruned and organized, and presented to the speaker.

Certain choices made by the Listener depend on the state of the knowledge-base at the time of the choice. The Listener checks old decisions to see if any new choices have become available and either displays this information to the speaker or places a notification on the agenda.

During command processing and clean-up, any conflicts that have arisen have been noted and saved. The last part of the clean-up process is to check for any contradictions that are still outstanding and try to correct them. The entire clean-up process is repeated until none of its stages changes the knowledge-base.

3.3.2 Library Processing

In order for the Listener to make use[29] of the cliché definitions:

- the clichés are compiled into a form the Listener can load
- the definitions are loaded, creating CAKE and Lisp representations of the cliché
- clichés are instantiated as requested by the Listener

Cliché definitions are compiled into a form usable by the Listener system. This process involves providing for the creation of both a Lisp representation of the clichés, which provides an index to information contained in the cliché, and a CAKE representation, which will cause the appropriate cliché reasoning to take place.

The cliché compiler implements the cliché-frame representation previously described. The top level cliché defining forms it processes are: Def-Cliche, Def-Relation, Def-Function, and Def-Action. There are some additional semantic details that were left out of the requirements cliché-frame description since, for the most part, they are included only due to idiosyncrasies of the implementation. These are:

[29]See files: RL, Relations, Conman, Objman, and Instman.

- Role definitions must include a **:Type** designator (after the role type) if the role can be filled by a CAKE type. This is due to specifics of the CAKE implementation.
- Cliché names automatically have a '%' character prefixed. This is necessary because CAKE does not support overloading of a name, e.g., a class cannot be called the same thing as a role. In retrospect, this problem and the :type problem should have been solved by a more sophisticated lexical front end.
- Roles can be annotated with information regarding when they should be printed out, i.e., always (the default), only in the document, only in the paraphrase, or never (for artifactual cliché roles).
- Roles can be designated as being important enough to merit querying about possible values when the role is unfilled (**:deduce-slot**).
- Consequents may use free variables in a limited fashion. The variables must be declared. e.g., **:Consequents (:Free-Vars ?Var)**. The form of a consequent that uses a free variable must be either: **(Implies (Member ?Var ...))** or **(Implies (Type ?Var) ...))**. The reason for the limitations is that a search procedure for potential instances of the free variable must be generated by the cliché library compiler, therefore some reasonable specification for possible values of the variable must exist, i.e., instances of a type or members of a set.
- Consequents can be propagated from a parent class to the definition of a child class using the **:Own-Consequents** keyword. The atom '!Self!' is used inside such a consequent to refer to the name of the child class.

The compilation of a cliché into a Lisp representation creates structures that represent the cliché and permit access to the components of its definition, e.g., its preconditions. Various cross-reference structures are also created, e.g., a list of all defined roles. The primary structures used during the running of the Listener are the various indexes used for informality resolution. These include the predictive index from role names to clichés and the specialization information used by the classification algorithm.

The CAKE representation of a cliché is more complex. Clichés are defined using CAKE objects: types, relations, and functions. For example, roles are defined as functions from the cliché type to the role type. Noticers are set up which cause the instantiation of the cliché definition when an object becomes an instance of the cliché. These noticers capture the cliché definition and perform the necessary substitution of the name of the particular instance being defined into the constraints. They also install the necessary dependency structure including default support premises. For consequents with free variables, the appropriate cascading sequence of noticers is set up to capture candidate instances of the free variable and to match them with instances of the cliché. For relations, noticers also handle the instantiation of defining-terms at the appropriate times.

These noticers also control the making of symbolic assertions. As discussed previously, for efficiency, cliché definitions are not instantiated on most symbolic instances. Generally, this has not caused a loss of deductive power since most symbolic references eventually acquire an object as a referent. In such a case, the definition is instantiated on the object and the semantics propagate from there. Certain symbolic instances do have cliché definitions instantiated on them. It turns out, for example, that certain cliché roles are unlikely to ever be equal to some object (e.g., a role that captures some qualitative notion and will generally only have constraints placed on it, but never a specific value). Other symbolic references represent instances that are named by a generator function applied to another instance, e.g., `(patrons university-library)`, that are also unlikely to ever be equal to some object. In these cases, the cliché definitions are instantiated on the symbolic instances. This is one example of the difficulty in deciding which objects and expressions are important to reason about and which are unimportant aliases.

Noticers are also created to detect and record new instances of a cliché for use by the paraphraser. Other noticers are set up to detect and record roles of objects that become equal to new values, again for use by the paraphraser.

The syntactic structure of constraints in cliché definitions is checked as the definitions are compiled. This procedure is used to detect potential errors in the cliché library. For example, the compiler will report if a predicate is used with multiple arities. It is also used to capture some of the basic domain structure and translate it into appropriate CAKE definitions. For example, new predicates are defined as CAKE relations of the appropriate type. Nevertheless, these checks are mainly for debugging and generally will not effect the runtime behavior of the Listener.

The Listener maintains a separate Lisp representation of the objects defined during a description session in addition to the representation stored in CAKE. The Listener represents these objects as frame instances and provides a simple interface which permits definition, type assignment, getting and setting of properties, access to parents, and access to instances. This representation is kept updated as new types and role values for an object are discovered by CAKE. The main use of this representation is to provide a definition of the structure of an object that can be used as a comprehensive description of that object. This is necessary, for example, since CAKE simply stores the values of roles without a representation of what is the valid set of roles for a particular object. Some external source has to supply information regarding applicable roles if a complete picture of a particular object is desired, e.g., during document production.

3.3.3 Informality Resolution

All assertions made to CAKE are passed through a procedure[30] that resolves informalities present in the statement and latent in the DKB. This procedure performs a number of checks:

- (re)checks the syntax of expressions and checks their arity
- ensures that each word used in an expression is known
- makes special checks on prefix expressions
- checks that expressions are defined or boolean based on their usage
- sets up classification and plausibility checks

The same syntactic checks used in the cliché library to check constraints statically are applied to expressions as they are asserted to CAKE. These checks are recursively applied to the arguments of the expression. The base of the recursion is the checking of individual words in the expression. Each word used in an expression must refer to either a cliché or an instance. If the word is unknown, a basic definition is created based on the arity of the expression the word is used in and whether or not the word is used as the head of a boolean expression.

Prefixes defined by the Listener system include: **`!decls`**, `!logic`, **`!set-of-all`**, **`!for-all*`**, **`!there-exists`**, and **`!if`**. The Listener implements a limited semantics for each of these, directed mainly at checking the expression for blatant errors. **`!decls`** is used to generate anonymous individuals that are used in expressions following the !decls statement. `!logic` is used to quote an assertion. Any free variables in the assertion are replaced with anonymous individuals and the resulting assertion is asserted. **`!set-of-all`** includes a predicate which defines the set of included elements. This predicate is instantiated with anonymous individuals and asserted (causing it to be run through the informality resolution mechanisms). No set theoretic semantics are implemented. The predicates in **`!there-exists`** and **`!for-all*`** are also instantiated with anonymous individuals and similarly the semantics of quantified logic are not implemented. **`!if`** has a predicate, then, and else clauses all three of which are instantiated and checked.

In order for CAKE to work correctly, assertions must be of type boolean and other expressions must be of type defined. It turns out that it can be difficult to arrange for expressions to be of the appropriate type. Therefore, to assure that the reasoning mechanisms work properly, all expressions are checked to see if they are of the appropriate type and if not a request is queued up to the clean-up procedure to assert them

[30]See files: Checkdef, Disambig, and Heuristic.

to be of that type. This can be thought of as a CAKE-specific informality resolution procedure. (For example, if the speaker asserts (informally) that `(implies A B)`, then he must mean for `A` and `B` to be boolean. If `A` or `B` is not boolean then CAKE will not necessarily deduce `B` if `A` is asserted.)

The mechanisms for assuring that expressions are checked for plausibility and classified correctly are driven by a class of assertions in CAKE of the form `(want-defined-exp operator !args arg1 arg2 ...)`. Every expression and sub-expression checked by the informality resolution process results in the assertion of a *want-defined-exp* for that expression (supported by the truth of the containing statement for that expression). For each true want-defined-exp, the Listener strives to ensure that a corresponding *defined-exp* becomes true. The plausibility checking mechanisms are responsible for seeing that the appropriate defined-exp assertions become true. (As the truth values of the want-defined-exp and defined-exp nodes change, a set of truth change noticers ensures that the appropriate processing is (re)invoked and the dependency trail ensures the proper support (or withdrawal of support) for results of the processing.) For example, asserting that an object is an instance of a cliché results in the assertion of defined-exps for each role of the cliché. Plausibility checking may, therefore, assert that an object is an instance of a cliché in order to bring about the appropriate defined-exp assertions. The want-defined-exp are also used on every command cycle as an index to the statements that are candidates for classification.

If the plausibility checking mechanisms are unable to find a way to cause a defined-exp assertion to be made for a corresponding want-defined-exp, then this task is placed on an agenda of issues to be resolved. The agenda entry includes a function closure that can be run to attempt to execute the task and a documentation string to assist the speaker when browsing the agenda. The speaker may run any agenda item to take action on it.

Certain expressions do not have want-defined-exp assertions made. These include expressions whose operators are prefixes or *universally defined* terms (e.g., CAKE built-in types). For efficiency, the implementation tries to avoid making want-defined-exp terms wherever possible. The mechanism described above is general and the dependencies are hooked up to achieve correctness in the face of non-monotonic reasoning. Unfortunately, this mechanism creates a large number of propositions and dependencies in the reasoning system and is thus potentially inefficient (especially relative to its function). A more efficient implementation might assume a monotonically increasing set of expressions (and sub-expressions). Of course, this could lead to belief in outdated facts. This tradeoff between correctness and efficiency is discussed more in Section 5.3.

Support for the Informality Resolution Mechanisms

For the most part, classification and plausibility checking are implemented in a straightforward manner in accordance with their description in the techniques section. Therefore, they need not be described again. There are some interesting mechanisms used to support these algorithms and they are described here.

All disambiguation decisions require assessing candidate object classifications. Each cliché has its preconditions stored on a property of its Lisp representation. Checking whether an object could be an instance of particular cliché involves retrieving the preconditions, making the appropriate instance and argument substitutions in the parameters of the precondition, creating the assertions corresponding to each precondition and querying the reasoning system for their truth values, and finally tabulating the results. The process can be terminated early if a precondition is found to be false. Otherwise, the candidates are rated and ordered and the calling process selects the best candidate.

This algorithm is potentially inefficient because each statement generates a candidate set of classifications for each true statement, each of which must be checked on every command cycle of the Listener system. The current implementation does in fact implement this in the obvious inefficient manner of rechecking all the appropriate preconditions on each cycle. This expense would become prohibitive in larger analyses ongoing for longer periods of time.

A more efficient implementation, which saves time for an additional memory cost, would be to use a Rete pattern matching network. This network would have to be incrementally updated at runtime as new instances became candidates for classification. The patterns to be detected would include the instantiated preconditions of various clichés that are possible candidate specializations for defined instances. The cost of establishing this network would be high initially but when used in a long ongoing analysis, in which most of the description tends to stabilize, the network would save the overhead of linearly checking each classification candidate repeatedly.

The *wish mechanism* is used to obtain appropriate truth values on propositions. It processes requests that propositions be made to have certain truth values, e.g., true or not-true. Contradictions can be resolved using this mechanism by processing a wish that a particular premise underlying the contradiction be *not-true*. Role values can be filled in by wishing that the proposition **(has-value (<role> <object>))** be true.

The wish mechanism does not use the brute force solution of simply asserting the desired truth value (which would not work for has-value assertions anyway). Rather it tries to bring about an evolution in the state of the knowledge-base that implies the desired truth values. Wish satisfaction is implemented recursively using a set of techniques that cause a gradual transition between the current truth value and the desired value. For example, to make a false node true it is first made not-false (usually unknown) and then true. The base case of the recursion includes: a routine for making an unknown node true, a routine for making a true (or false) node unknown, and a routine for retracting underlying premises. Wishes that cannot be satisfied are put on the agenda.

The routine for making an unknown node true can only handle specific kinds of nodes (since it avoids the brute force technique of asserting the node). Currently, it can only handle making ***has-value*** nodes true. (Has-value nodes are defined in CAKE to be true if and only if their argument is currently equal to a value term.) This requires searching for a value for the argument of the has-value node. The idea is basically the same as searching for alternate values in an equality except there is no initial value.

The routine for making a true node unknown implements the premise altering heuristics described previously. Briefly, this routine searches for alternate values in an equality, searches for sibling types in a type assertion, or recursively applies these techniques to a default.

Any choices made in satisfying a wish are continually rechecked for additional information. This is achieved by silently rerunning the routine that eventually achieved the correction and observing changes in resolution choices.

3.3.4 Paraphrasing

In the implementation[31], detection of interesting deductions serves to provide both the command paraphrase and the feedback of interesting deductions. This works since the command language is based on a entity-relationship model. The set of interesting deductions corresponds to the kinds of statements that are natural to make in such a model, e.g., declaring the type of an object or filling in a role. The set of interesting deductions naturally contains the basic implications (paraphrase) of input statements.

During processing, observations are queued up for presentation during clean-up. Nothing is printed directly since it may only be transiently true. The events that are observed are: the creation of a new requirement, entering a context, creation of a new object, loss of a type (on an object or a role), gaining of a type, losing a role value, gaining a role value, objects with equal definitions, and structural changes to clichés (adding a cliché or adding a role to an existing one).

The observations that are queued up are pruned and then presented to the speaker. Each kind of observation has an *:includer* and a *:presenter*. Includers determine whether an observation is still timely and if so whether it should be presented given the other observations. Some examples include: a lost value is not included when a

[31]See file: Present.

replacement value is also observed, only the most specific observed type is reported. and role types are not reported if the role has a value. Includers have access to all observations that are candidates for presentation.

The pruned observations are indexed about their subject objects, e.g., the object whose role has gotten the value. The groups of observations regarding subject objects are presented in order of the definition of the subject objects. The presenters decide how to display the observations that pass the test of the includers. Observations regarding new or lost values for multi-valued roles are grouped and printed together. Types and slots can be annotated as being internal (e.g., artifacts of the construction of the cliché library) and the presenters will ignore them. The presenters also ensure that the appropriate synonyms are used in the output along with the appropriate quoting of new terms.

3.3.5 Contradiction Handling

CAKE handles contradictions through the use of a *backtracker*. The backtracker is invoked whenever CAKE discovers a contradictory clause. The default action of the backtracker is to interrupt processing and invoke the *premise selector* to solicit input regarding a premise to retract.

The Listener redefines the backtracker[32] to defer contradiction handling and simply collect contradictions as they arrive. After an input command is processed, a routine to resolve all pending contradictions is invoked. Each contradiction is analyzed by designating its supporting premises either interesting or uninteresting. This analysis divides premises according to the following classification:

- rules - premises that begin with 'implies' and that are marked as coming from clichés are part of the cliché definition and considered uninteresting because cliché definitions are assumed to be correct. Any errors are attributed to the selection of the wrong cliché.
- defaults - are interesting because the speaker may want to change them.
- choices - are interesting because the speaker may want to change them. The speaker and the Listener both make many choices. The speaker needs to be able to change the Listener's choices in case the Listener's heuristics yield an incorrect result.
- equality assertions - generally correspond to roles that have been filled and are interesting because new role values might have to be specified.

[32]See file: Contras.

- cliché version markers - are uninteresting. They are an internal premise class that indicate which version of a cliché definition is active. This permits cliché redefinition within a session.
- type assertions - are interesting since they represent the classification of an object.
- other assertions - are interesting as a matter of completeness.

The contradiction with the smallest number of interesting premises is chosen as the one to resolve first. The Listener helps perform the resolution by invoking an augmented premise selector. First, the contradiction is explained by printing out each node involved, a short documentation string, and interesting premises in the following order: equalities, type assertions, other assertions, defaults, and choices. This weakly reflects the likelihood of the premise needing to be retracted to fix the contradiction with the most likely candidates displayed first. The speaker is given the opportunity to retract one of the premises. If no premises are retracted, the speaker is given the opportunity to invoke the premise altering heuristics. Successful application of one of these heuristics, substituting a related premise with equal information content, makes the target premise untrue and resolves the contradiction.

3.3.6 Reasoning System

The Listener interacts with CAKE through an extra layer of interface defined in the Listener code[33]. This interface provides a recording mechanism for tracking all interaction with CAKE. Each command statement to the Listener begins and ends an interaction transaction with CAKE. Beginning a transaction creates the statement support premise which is used as the support for all assertions made as part of the translation of that statement. Initially, the transaction mechanism was intended to assist in debugging and it proved useful for understanding the content of the interactions the Listener had with CAKE. Currently, it is simply providing a count of the number of interactions with the reasoning system. This count is revealing in that it shows how complicated a seemingly straight-forward statement can be. Some statements require over 400 interactions with CAKE to complete their processing. The recording mechanism is also tracking all facts asserted as premises. This set of premises can be used in the validation of the description.

The interface provides routines that echo CAKE's own routines for asserting, denying, and retracting propositions. The interface routines provide some additional checking and functionality and also simplify the interaction (for this application)

[33]See files: Cake and Check.

with CAKE. The assertion routine can make an expression a premise, an axiom, or supported by other assertions. If support is provided, the routine checks, as a debugging step, that the support is true (i.e., if the support is not yet true then the assertion should be made as an implication between the support and the constraint). If the asserted expression is a premise and is tautologous, the speaker is informed of the redundancy. The theory behind reporting tautologies is that it is an indication that the speaker has not realized the full import of his previous statements and therefore it is worthy of notification.

The most critical purpose of this interface is that it provides the point at which the processing to check for informalities can be hooked in. By default, all assertions that are made through this mechanism are checked using the informality resolution mechanisms described above.

A major initial goal of this interface, that proved unnecessary, was to micro-manage the interface to the reasoning system. Each interaction with CAKE has an expected outcome, e.g., an assertion is expected to result in the proposition being true and a retraction to result in the truth value being unknown. But these expectation do not always materialize, e.g., contradictions occur or an unknown node may unexpectedly become true if you *ask* about its truth value. Micro-managing the interface involves anticipating all the possible outcomes and handling the various situations. The problem with micro-managing is that at this most primitive level of interaction there is no data available to help fix problems. Also, there are really only two outcomes of an interaction with CAKE, success or contradiction. Therefore, correction of problems is left to the contradiction handling mechanisms.

Augmentations to CAKE's Reasoning

A number of modifications[34] and augmentations were made to CAKE's reasoning mechanisms.

CAKE will normally attempt a proof-by-contradiction when asked about the truth value of a currently unknown node. This had to be turned off because it was creating excessive load on the system. Proof-by-contradiction temporarily asserts the truth of certain propositions, i.e., it creates a hypothetical world. The Listener would pursue the implications of these hypothetical worlds, instantiating many clichés and creating large numbers of terms, only to have the results unused when the assertion was retracted. The number of hypothetical worlds proved to be too great for the current technology. Viewing proof-by-contradiction as the creation of hypothetical worlds may provide insight to those using logic to reason about and control representations outside of the reasoning system.

In CAKE, certain terms may be designated as values. Each equality class may

[34]See files: Cakepatch, RLini, Values, Notice, Hvaluel, and Watcheq.

contain at most one value. Sometimes, however, that value cannot be determined, even though a number of *good names* are known for elements of the equivalence class. Good names are useful to announce in the feedback provided by the Listener when they are the best description available. Adding the concept of good names to CAKE required some modifications (described below).

A CAKE expression may be a symbolic reference, e.g., the variable `X`, the role `(prop-1 obj-1)`, or the function `(F X)`. An expression may also be a good name for an object (or set of objects), e.g., an expressive intentional description whose extension is difficult to determine such as "the set of all people with brown hair" or a lambda expression, `(Lambda (X) (1+ X))`, which represents an infinite class of equivalent lambda expressions. A good name conveys information about the class of objects. Finally, an expression may be a value that denotes a unique individual in the world, e.g., the number 5, or the person George-Washington.

An object may be referred to by many symbolic references, many good names, but only by a single value. These distinctions have interesting ramifications in the CAKE's reasoning processes. In reasoning with equalities, one does not want to perform all reasoning using every member of the equality class. In general, this duplicates a lot of work and is very inefficient. CAKE stores equivalence classes as trees and applies its reasoning process to selected representative elements of the class without loss of generality or reasoning power. Thus, reasoning cannot depend on syntactic properties of an expression. Adding the notion of good names (and requiring special reasoning on value and good name terms) forces a consideration of syntactic properties of expressions.

The top of the tree that represents an equality class is a value, whenever one exists for the class. The tree is kept as a balanced tree and reasoning processes (e.g., noticers) are normally applied only to the expression (which may or may not be the value term) that triggers the reasoning. Provision is made (in the form of image noticers) to cause noticers to be applied to an expression's parent (and recursively to the parent's parent) in the equality class (each expression has only one parent). This algorithm guarantees that processing is applied to the root of the tree, i.e., the value of the class. This allows special reasoning on value terms while keeping the number of expressions to which the reasoning is applied reasonably small. For example, in a tree of breadth 3, and depth 2, starting from a leaf, the reasoning processes are only applied to 3 of a possible 13 expressions.

In order to add the notion of good names to CAKE, the method of creating the equality class trees was altered to put good names at the top of the class when a value does not exist. Since an expression can have many good names, a metric was defined which allowed CAKE to put the best good name at the top. This extends the guarantee that image noticers will be applied to value representatives of an equality class to ensure that noticers will be applied to the best name of an equality class when the value is unknown. This guarantee is necessary to implement a number of Listener functions, paraphrasing in particular.

To phrase the discussion specifically in CAKE implementation terms: CAKE makes certain guarantees of reasoning completeness regarding the invocation of transparent noticers. Transparent means that the noticer function does not make any tests in its body that rely on the syntax of the pattern that triggered it [37]. The Listener implements a loosening of the definition of transparency. Noticers are allowed to rely on a limited property of the syntax of the triggering pattern, i.e., it may test whether or not the term is a value or a good name and use that value or name.

The paraphrasing mechanism makes use of the guarantee regarding values and best names being at the top of the equality class tree and therefore having image noticers run on them. One of the jobs of the paraphrasing mechanism is to notice when expressions become equal to values (or best names) and to announce this new equality. Since image noticers are always run on the expression at the top of the equality class tree, then announcing new values can be implemented by a noticer on the term **(has-value <expression>)**. When the expression gets a value, the noticer will be called on **(has-value <value>)** (which will be true) and with the proper indexing this can serve to announce an expression getting a value. A similar sort of mechanism is implemented for multi-value equivalences, e.g., **(member 3 A)**. The added difficulty in multi-valued equivalences is simply that they may be cascaded with single value equivalences, e.g., **(member x A)** and **(= x 4)**.

CAKE's equality reasoning is supplemented with mechanisms to support equality-views. When an equality-view is defined for a cliché, a CAKE frame type is defined that has slots corresponding to the roles designated in the equality view. Whenever an instance of the cliché is created, a corresponding frame instance is created for any equality views (by applying an overlay to the instance that returns a frame instance). Equalities are checked for, in a relatively inefficient manner, by asking about the pairwise equality between new frame instances and old ones. When an equality is detected, the appropriate reports are queued up for observation.

3.3.7 Document Generation

The requirements document is generated by a two phase process[35]. The first phase is responsible for determining the content of the document. The second phase is responsible for producing the text (or other appropriate presentation) of the document. The first phase executes a structured walk-through of the RKB deciding what to describe and placing information into a document parse tree. The second phase walks over

[35]See file: Docman.

this parse tree and generates intelligible output. In this case, a LaTex source file is created. A third phase responsible for producing aesthetic output would also be useful. The third phase could perform source to source transformations on the output of the second phase. Some constraints on document appearance can be most naturally stated as context sensitive rules of document transformation. Such constraints can be enforced in the first or second phase, but this adds complexity to the generation procedures. The issue is a tradeoff between using a procedure that generates only correct answers and using a partially correct generator with a tester (and debugger). Other phases might be useful to handle different kinds of constraints on the output, e.g., grammar, spelling, syntax, and presentation order [34].

In creating the document parse tree, the first phase uses the following primitives to add content to the document.

- text - adds text to the document
- bold-text - adds emphasized text
- text-raw - adds text and skips phase 2 processing
- chapter, section, subsection - starts one of these textual units
- paragraph - starts a paragraph
- newpage - starts a new page
- vspace - adds some vertical space
- spacer -adds a heading as space between objects
- itemize - adds an itemized list

(Note: LaTex [40] is a text formatter in the family of Knuth's Tex formatting programs. The details of its capabilities are unimportant to understanding the document generation process. Phase 2 routines encode knowledge about specific LaTex commands used to produce certain document features, e.g., section headers. Phase 1 routines rely on Phase 2 supporting a certain generic interface which includes the concepts described above. The output model of phase 1 is not completely generic and strong influences from LaTex can be seen.)

The document consists of: a title page, table of contents, introduction, environment, needs and system chapters, and the requirement's state chapter. A separate reader's guide is also provide to assist in reading the document. The environment, needs, and system chapters each include a dictionary of terms used in that section. Each of these will be described in more detail below. This decomposition defines the basic, fixed, top level structure of a requirements document. Additional structure may be added within this framework during creation of the document.

The reader's guide is a condensed version of the description presented here. It provides insight into the procedure used to generate the requirements document. Since the document is generated by a fixed algorithm, each sentence has a precision that aids in its interpretation. For example, syntax conventions are used consistently. The presence or absence of a description conveys information since the document is created by a predictable process.

The title page is generated by a simple phase 1 procedure that extracts the name of the requirement from the RKB and the current time from the host computer system. LaTex provides the capability to generate a well formatted title page, thus phase 2 simply generates appropriate LaTex commands. Phase 2 also generates a LaTex preamble which controls parameters such as document style, type size, paragraph indenting, margins, and fill mode.

The table of contents is generated by LaTex when the appropriate section heading commands are used. No phase 1 processing is required specifically for its generation. The only required phase 2 processing is to insert the command that requests printing of the table of contents.

The introduction is generated by creating a top level description of the environment and system objects based on the clichés they are instances of. Each cliché includes various levels of canned-text documentation, among them are the *abstract* and *overview* levels. The abstract level is used in generating material for the introduction, whereas the overview level is used in succeeding, more detailed, descriptions in the following chapters.

The environment section and the system section are created by the same mechanism. Environments and systems are basically the same kind of things. It is mainly perspective that differentiates them. This can be seen by considering the following. One requirement's system may be another's environment. For example, banking environments have required ATM systems to be developed. The ATM system is the environment for the ATM software system.

The environment section starts with a detailed description of environment objects. This description begins with the overview level of canned text associated with the most specific types of an environment object. Then the slots of the object are enumerated and described. If a slot has a value, then this value is presented. If the slot has no value, but it has a type, then the slot is documented as an instance of the appropriate type. If a slot has neither a value or type, it appears in the list of unfilled slots.

Certain slots on particular clichés are designated as worthy of description in separate sections. For example, on the high level clichés *environment* and *system* there are slots that hold information about actions, agents, and behaviors that are printed in separate sections. Document sections can be added on the fly and fit into the basic fixed top level document structure.

Each of the environment, needs, and system sections contains a dictionary. These dictionaries describe technical terms used in the section. Certain technical terms (those defined in a reference library of clichés), e.g., system or need, are not included in the dictionary under the assumption that they will be documented in a user's manual.

The needs section includes a description of all the needs (problems to be solved, policies to be enforced, and product functions to be supported) that the system has to address. The needs define the problem in the environment that the system to be built has to solve. The method of description is basically the same as in the environment and systems section. Needs objects tend to be less structured, however, therefore the descriptions are generally shorter.

The requirement's state chapter describes the loose ends in the current state of the requirements acquisition. The loose ends include:

- any significant true facts that did not make their way into the document. These are discovered by searching through the knowledge-base for assertions of a certain syntactic form (excluded are type assertions and equalities which will have been included in the document) that would not have been included in the requirements document due to the nature of the object-type-role centered traversal of the RKB.
- a list of retracted defaults and other retracted decisions
- tl ` pending agenda items
- a list of roles that were queried about in creating the document that did not have values
- any outstanding contradictions

Chapter 4

A Listener for Legal Description

This section describes an example run through the Listener system using the domain of legal document descriptions. The basic idea is that the speaker (a junior law associate) describes the conditions and background of a legal document (e.g., a will, lease, or employment agreement) and the Listener attempts to fill in details, resolve ambiguities, and detect contradictions. The legal domain was chosen because of its well codified technical vocabulary and conceptual structure, one of the important prerequisites for Listener applications.

In comparison with the requirements examples, the legal example uses the same basic capabilities and techniques, the same domain modeling technology (a cliché library in the domain of legal documents), but a slightly different document generator. A different document generator was necessary since there is some procedural embedding in the RA document generator particular to the domain of requirements acquisition. This legal document generator is also slightly different in that it generates the actual legal document as opposed to a description of the document. (In analogy to the requirements document, one could have chosen to generate the requirements of the legal document versus the actual document, but these two concepts are very close, much closer than a software requirement and the actual software. Therefore, legal boiler-plate was installed as the canned text, permitting generation of an actual document.) A more general document generation facility, without any procedural embedding, would be useful to create. This would require a general theory of document structure and content so that clichés (e.g., the Requirement or Legal-Document clichés) could be annotated with a declarative description of the document structure. The entire text of the generated contract is included in this section, beginning on page 139, after the legal scenario. Note that it took less than ten days to implement this example, including the time to write the new document generator. The bulk of the time was taken in writing the new cliché library, for which there is currently no support.

Figure 4-1 depicts the clichés defined to support this example. Two trees of clichés are shown. Legal documents are defined, including *contracts* and *employment-agreements*. The scenario focuses on a particular kind of employment agreement. For breadth, the clichés *will*, *deed*, and *purchase-and-sale-agreement* are also defined. The second tree, with root *provision*, defines provisions of contracts, in this case of employment agreements. Typical provisions, i.e., concerns, include maintaining the confidentiality of provided data, rules for premature termination of the contract, and agreement as to who owns the intellectual rights to work produced under the contract. The legal scenario, making use of this library, follows:

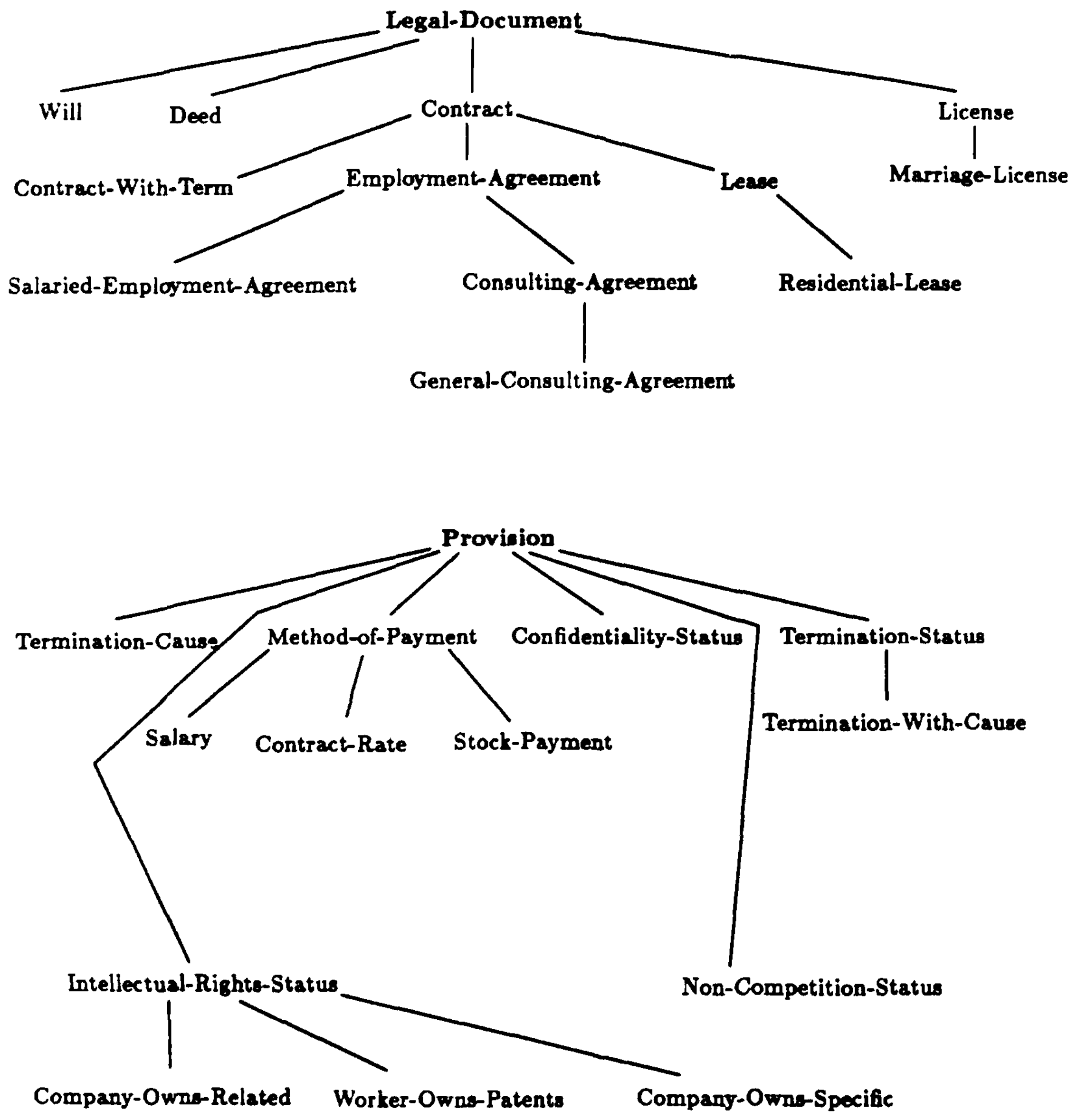

Figure 4-1: Legal Cliché Library

```
1: (Find Legal-Document Connor-Contract :Contract :Synonym Cc)
Beginning-A-New Legal-Document Called-The Cc.
Cc Is-An-Instance-Of Contract.

2: (Define Connor-Software-Associates :Company :Synonym Csa)
Csa Is-An-Instance-Of Company.

3: (Fill-Roles Cc :Offering-Party Csa :Accepting-Party Rudolph-Glikin)
Cc.Accepting-Party Has-Value ''Rudolph-Glikin''.
Cc.Offering-Party Has-Value Csa.

4: (Fill-Role Cc.Remuneration.Daily-Rate With 1000)
Connor-Contract.Remuneration Is-An-Instance-Of Per-Day-Payment.
Connor-Contract.Remuneration.Daily-Rate Has-Value 1000.
```

Figure 4-2: A Contract Document

The scenario begins, as do the previous examples, with setting up the basics. Command 1 gives a full name and short name to the legal document that will be described. It states that the document is also a specialization of a contract (versus, e.g., a will or deed). Command 2 identifies one of the agents participating in the contract as a company with the short name "CSA." In this domain, one use of short names (and of the most specific type deduced for an agent) is in generating the part of the document which follows the template: <long-official-title-of-agent (hereinafter referred to as short-name)>.

Command 3 simply states who the contract is between. Command 4 defines part of the payment involved in the contract, but at this point the contract could still be a lease or an employment agreement. The classification mechanism cannot yet make a specific deduction. However, as encoded in the cliché library, it cannot be a purchase and sale agreement due to the periodic kind of payment.

```
5: (Fill-Role Cc.Remuneration.Stock-Vested-Date With Cc.Termination-Date)
Cc Is-An-Instance-Of Salaried-Employment-Agreement.
Cc Is-An-Instance-Of Contract-With-Term.
...
6: (Fill-Roles Cc :Effective-Date 4/20/90 :Term (Years 2))
Cc.Effective-Date Has-Value ''4/20/90''.
...
7: (Define Cc.Work-Products :Advice
    :Roles (Topic Artificial-Intelligence-&-Software-Engineering))
Cc Is-An-Instance-Of General-Consulting-Agreement.
Cc.Intellectual-Rights Is-An-Instance-Of Company-Owns-Related.
...
```

Figure 4-3: A General Consulting Agreement

Command 5 continues to expand on how payment is to be made. The fact that stock is included allows the Listener to exclude the possibility of the contract being a lease which leaves salaried employment agreement as the only available option. Further, employment agreements are contracts with terms, i.e., non-instantaneous and of finite duration.

Command 6 defines the term of the agreement (and the start date). Command 7 indicates that the work to be done includes giving advice about artificial intelligence and software engineering topics. These statements allow the Listener to conclude that the contract is a specialization of employment agreements, i.e., a general consulting agreement. In such an agreement, as in contracts in general, there are a number of provisions (which are indicated by numbered paragraphs in the contract). One of the provisions regards ownership of the intellectual rights to work produced under the agreement and in general consulting agreements it is assumed to default to the company that hired the contractor owning all rights related to the work hired for.

```
8: (Define Cc.Non-Competition :Roles (Causal-Duration (Years 1)))
Cc.Confidentiality Is-An-Instance-Of Termination-Cause.
Cc.Intellectual-Rights Is-An-Instance-Of Termination-Cause.
Cc.Termination-Conditions Is-An-Instance-Of Termination-For-Cause.
Connor-Contract.Non-Competition Is-An-Instance-Of
                For-Cause-Non-Competition-Status.
Connor-Contract.Non-Competition.Causal-Additional-Time Has-Value (Years -1).
Connor-Contract.Non-Competition.Causal-End Has-Value 4/20/91.
...
There-Are 1 Pending-Contras
A-Contradiction-From-A Negation Clause-Between->
(Date>= 4/20/91 4/20/92)
(Not (Date>= 4/20/91 4/20/92))
The-Interesting-Premises-Are
2. (User= (Effective-Date Connor-Contract) 4/20/90)
3. (User= (Daily-Rate (Remuneration Connor-Contract)) 1000)
4. (User= (Termination-Date Connor-Contract)
          (Stock-Vested-Date (Remuneration Connor-Contract)))
6. (User= (Causal-Duration Connor-Contract.Non-Competition) (Years 1))
...
Would you like to retract a contra premise?(Y or N) No.
Would-You-Like-Suggestions-For-Alternate-Premises (0 is no, # or list of #s) 6
In-The-Premise
 (User= (Causal-Duration Connor-Contract.Non-Competition) (Years 1))
You-Could-Make-A-Substitution-For-The-Word (Years 1)
Choose another number of Years 3
Resolved-Contradiction
Connor-Contract.Non-Competition.Causal-End Has-Value 4/20/93.
Connor-Contract.Non-Competition.Causal-Duration Has-Value (Years 3).
...
```

Figure 4-4: A Time Conflict

In command 8, the non-competition provision is discussed. The causal-duration is the length of time of the non-competition clause when the contract is terminated for cause. The termination-conditions of the agreement become an instance of termination-for-cause. Cause is initially defined in a consulting agreement to be breach of the confidentiality provision, breach of the intellectual-rights provision (both are instances of termination-cause), or breach of the non-competition provision.

This command causes a contradiction, however, in that the duration of the contract is longer than the duration of the penalty non-competition time (i.e., the causal-additional-time equals -1). The mistake is simply that causal-duration is relative to the contract effective date, not the contract termination date. The problem is fixed by making the causal-duration 3 years (1 plus the contract term).

```
9: (Define Cc.Intellectual-Rights :Worker-Owns-Copyrights)
...
There-Are 1 Pending-Contras
A-Contradiction-From-A Disjoint Predicates Clause-Between->
(Company-Owns-Related Connor-Contract.Intellectual-Rights)
(Worker-Owns-Copyrights Connor-Contract.Intellectual-Rights)
...
Would you like to retract a contra premise?(Y or N) No.
Would-You-Like-Suggestions-For-Alternate-Premises (0 is no, # or list of #s) 10
Default
 (!Default 1 (Company-Owns-Related (Intellectual-Rights Connor-Contract)))
Can-Be-Retracted With-The-Following-Replacement-Options-For-Supported-Premise
 (Worker-Owns-Patents Worker-Owns-Copyrights Company-Owns-Specific
  Company-Owns-Related-Except-Scheduled Intellectual-Rights-Status)
Should I retract default?(Y or N) Yes.
Possible-Replacement-Types-In-Type-Assertion
(Not (Company-Owns-Related (Intellectual-Rights Connor-Contract))) Are
Which-Should-I-Choose as a :Type-For-Replacement
 (Worker-Owns-Patents Worker-Owns-Copyrights Company-Owns-Specific
  Company-Owns-Related-Except-Scheduled Intellectual-Rights-Status)
(1 based, 0 for none) 4
Resolved-Contradiction
There-Are-Still-Pending-Contras
...
The-Interesting-Premises-Are
(Number-In-Parens-Greater-Than-One-Is-Contras-Premise-Is-In)
...
2. (Worker-Owns-Copyrights Connor-Contract.Intellectual-Rights)
3. (!Choice Raap 9 (Company-Owns-Related-Except-Scheduled
                    (Intellectual-Rights Connor-Contract)))
Would you like to retract a contra premise?(Y or N) Yes.
Retract premise # (0 Is None and Loops to all)? 2
Resolved-Contradiction
Connor-Contract.Intellectual-Rights
                Is-An-Instance-Of Company-Owns-Related-Except-Scheduled.
```

Figure 4-5: An Intellectual Rights Conflict

In command 9 the associate states that the worker wishes to retain ownership of the copyrights to the work performed. This is in direct contradiction with the company owning the rights to all work products produced under the contract. The contradiction is explained and the underlying premises are presented. The associate chooses to retract and alter the default assumption that the company owns all related rights and chooses as a substitute, and a compromise, that the company will own all related rights except for rights in technology that is currently under development by the consultant as listed in an attached scheduled. This allows the consultant to preserve rights in technology he is developing to assist in a successful consulting business. The compromise is completed when the contradiction with the previous assertion, regarding owning all copyrights, is fixed by retracting that previous assertion.

The scenario concludes here. The next four pages reproduce the document produced by the Listener. Some interesting points about the contract include:

- Numbered paragraph headers in bold font correspond to individual provisions (roles) of the contract. Provisions that are addressed solely by common law are not included in the document, but are enumerated in the final page (not actually part of the contract) as part of the document state description. The cliché hierarchy actually includes a cliché called *common-law*. All *direct specialization* of common-law define the common-law for a particular provision of the contract. A provision is not included in the written document unless it is an instance of a cliché two levels below common-law, i.e., it is not just common law but an elaboration of the common law.

- Lines in the document followed by a word in parenthesis indicate unfilled roles. In this way, the document is a template for a contract the can be completed by filling in the blanks. Some details to be filled in include: the number of shares of stock involved and, of course, the required signatures.

- The final page of the print out is not part of the contract. It is a summary of the state of the description. This includes an enumeration of relevant common law provisions (that are implicitly included) and a list of unfilled roles.

- The document generator permits cross-references to various provisions in the document (e.g., see paragraph 7, item 3).

Connor-Software-Associates General-Consulting-Agreement

This Agreement effective as of 4/20/90, by and between Connor-Software-Associates of ____________ (address) (hereinafter referred to as ''CSA'') and Rudolph-Glikin of ____________ (address) (hereinafter referred to as ''Consultant'') as follows:

1. **Period of Performance.** Consultant shall provide services hereunder for a term of (Years 2) beginning 4/20/90 and ending 4/20/92 (the ''Term'').

2. **Scope of Work.** Consultant will provide advice, counsel and expert opinions in the area of, but not limited to, Artificial-Intelligence-&-Software-Engineering and any other work reasonably requested by CSA that is related thereto. Consultant will provide such services at a minimum amount of One-Day-Per-Month which may increased by mutual agreement up to a maximum of One-Week-Per-Month.

3. **Payment.** As consideration for the services to be provided hereunder, the CSA agrees to pay *Consultant 1000 dollars per day.*

In addition, ____________ (number-of-shares) of Common Stock of CSA will be vested to Consultant on 4/20/92, as set forth in a Stock Agreement to be executed by the parties at the same time that this agreement is executed. A copy of the fully executed Stock Agreement is attached hereto.

4. **Confidentiality.** Consultant acknowledges that information not generally known outside of CSA concerning or related to the research, design, development, production and sale of products, including without limitation the general business operation of CSA, the findings, reports, inventions, discoveries, developments and improvements disclosed to Consultant by CSA or written, invented, made or conceived by Consultant under this Agreement, is confidential and of great value to CSA. Consultant agrees not to divulge to anyone, either during or for a (Years 3) period after the Term, any such information obtained or developed by Consultant during the Term.

5. **D overies, Inventions, and Copyrights.** Consultant shall specifically describe and identify in an addendum to this agreement all technology and inventions that Consultant has rights, interests in, or has under development, that may be used in providing the services hereunder, and which rights or interests Consultant has an interest in protecting. All ideas, findings, reports, invention, writings, disclosures, discoveries, developments and improvements (hereinafter the ''Work

Products'') written, invented, made or conceived by Consultant in the course of or arising out of the services to be performed hereunder (except those so scheduled), whether or not reduced to writing or practice, shall become and remain the sole and exclusive property of CSA.
Consultant hereby transfers and assigns to CSA all right, title and interest in and to the same whether or not patent applications are filed thereon. Consultant agrees to execute any documents and to cooperate with CSA, at the expense of CSA, in any actions deemed necessary by CSA to secure fully to CSA all rights in such Work Products. All software, data, documentation, memoranda, papers, drawings, reports, photographs, audio and audio visual works or materials of any kind or media produced hereunder shall be considered work for hire and Consultant hereby waves any claim or copyright therein.
Upon the expiration or earlier termination of the Term, Consultant shall promptly deliver to CSA all documents, papers, drawings, software, specification, reports, and materials of any sort which were furnished by CSA to Consultant or which were prepared by Consultant in performance of the services hereunder.

6. Non-Competition. During the (Years 2) Term of the agreement, or for (Years 3) in the event of the termination of Consultant for cause as defined in Paragraph 7 below, Consultant agrees that he shall not, for any cause whatsoever, directly, indirectly or otherwise engage on his own account in, or enter into the employ of, or render any service whatsoever to, or own any interest in, any other person, partnership, association, corporation, or other organization which is directly competitive with the business of CSA anywhere in the United States, except with the prior written consent of CSA; notwithstanding the foregoing, the Consultant may own shares (not exceeding 2 percent of the outstanding shares) in a corporation whose shares are traded on a national stock exchange or over the counter.
Consultant further agrees that during the Term and for a period of (Years 2) after its expiration or termination he will not:

(1) hire, offer to hire, entice away or in any other manner persuade or attempt to persuade any officer, employee or agent of CSA to discontinue his or her relationship with CSA without permission of CSA;

(2) directly or indirectly divert or take away, or attempt to solicit, divert or take away any business CSA had enjoyed or solicited during the Term; or

(3) assist any member of his immediate family or any business associate to

commic any of such acts within the territory herein described.

If any part of this Paragraph 6 should be determined unreasonable in duration or area, then this Agreement is intended to and shall extend only for such a period of time and in such area as determined to be reasonable.

7. Termination. CSA or Consultant may terminate the this Agreement in the event of the breach by the other party of any term or condition of this Agreement upon ten (10) days written notice to the breaching party.
The Agreement shall be deemed to have been terminated for ''cause'' for purposes of this agreement if CSA terminates the Consultant for any of the following reasons:

(1) commission of an illegal act, including theft or embezzlement;

(2) substantial failure of Consultant to perform the services that he is obligated to perform hereunder in a timely manner, for any reason other than death or disability; or

(3) breach by Consultant of any of the provisions of Paragraphs 4, 5, or 6 hereof.

8. Jurisdiction. This agreement shall be governed by the laws of ____________ (state). If any provision of this Agreement shall be held invalid or unenforceable by a court of competent jurisdiction, such provision shall be deemed omitted to the extent required by such judgment and the remainder of the Agreement shall continue in full force and effect.

IN WITNESS WHEREOF, the parties hereto have executed this Agreement as of the day and year first above written.

Consultant	Connor-Software-Associates
X ______________________	X ______________________
Date ______________________	Date ______________________

```
The following provisions while not explicitly included in the contract are
addressed by Common Law:

The common law regarding modification of the contract generally requires
that changes be mutually agreed upon in writing by all parties involved.

The common law regarding notices generally requires that any notices
required under the terms of the contract be in writing and delivered in
person or via registered U.S. mail.

The following information is missing from the document:

   • State of Connor-Contract.Jurisdiction

   • Number-of-shares of Connor-Contract.Remuneration

   • Address of Rudolph-Glikin

   • Address of Connor-Software-Associates
```

Chapter 5

Issues and Observations

Most of the exposition, until this point, has concentrated on a discussion of how the Listener works. This section explores some issues in the design and use of the Listener system. First the crucial topic of domain modeling is discussed and some insight is provided into the issues of constructing a cliché library. This includes a discussion on the topic of reuse and how well the Listener facilitates reuse. Next the utility of clichés for capturing shared knowledge is discussed. Then the breadth of the informality resolution techniques is discussed. Finally, a discussion of some of the issues of interfacing the RA with a design assistant is presented.

The history of the research project is briefly discussed in the following paragraphs. Work proceeded in a spiral fashion with greater attention alternately being payed to the various work products, i.e., the scenario, the document to be generated, and the cliché library.

The initial Listener proposals (early 1987) [41, 42] use the library database problem as a point of departure. They contain some document fragments showing what a requirement produced by the RA might look like, a discussion of what some of the clichés in the cliché library should be, and a proposed scenario illustrating how the Listener might be able to assist a requirements analyst.

Initial work on the Listener focused directly on supporting the proposed library scenario. However, it gradually became apparent that it was not possible to support the scenario, nor even have a clear idea of the exact content of the scenario interaction, without having a clear idea of what the end product of the requirements analysis should be. In cognizance of this, the focus of research was temporarily switched to a detailed investigation of the contents of the requirement.

This second stage of research culminated in early 1988 with a concrete proposal for the requirements document to be produced by the RA for the library database problem. More importantly, it culminated in a general theory for how the requirement should be structured, i.e., a requirement has three main sections: *environment*, *needs*, and *system*. This improved understanding of the content of the requirements document and its organization facilitated the creation of an improved cliché library.

Armed with a concrete requirement as the goal, the focus of research returned to implementation of the RA. First, the scenario was revamped so as to provide (or access in the cliché library) the information content contained in the proposed requirements document. By early 1989 the scenario was completely implemented. Soon thereafter, the document generation capability was completed.

In the fall of 1989 work was begun, by Paul Lefelhocz, on the air traffic control scenario. This work was completed in early 1990 and is described in section 2.10 and in [28]. In the spring of 1990, the author completed the legal domain example described in section 4. Each of these examples was approached by first creating a plausible scenario for the Listener to execute, then coding a cliché library with enough depth to support the scenario and enough breadth so as not to be trivial. Finally, the capabilities of the Listener were used to execute the scenario. These capabilities transferred well between examples and after writing the cliché libraries the implementation of the scenarios proceeded rather smoothly.

One valuable experiment that has not been done with the Listener is to create a comprehensive cliché library and then see how well that library supports various acquisition scenarios. Such an experiment is necessary to validate the true usefulness of this technology. It should be noted, however, that the underlying motivation of this project was to demonstrate the interesting kinds of behaviors that could be supported by an automated Listener system in order to provide motivation (and a representation technology) for the creation of large cliché libraries. In addition, supporting more realistic acquisition scenarios would require development of the system from the demonstration stage to at least the prototype stage. The capabilities of the Listener have been demonstrated to be useful. It remains for future work to build on this in a more realistic setting.

5.1 Domain Modeling

How well can the world be modeled using clichés? How effectively can clichés be accessed and computed with? The answer to these questions depends to a great extent on how easy it is to capture knowledge in independent self-contained units, how these units relate to and communicate with each other, and how easy it is to access and reuse these units. The question of representational adequacy of the cliché-frame representation for individual cliché units is left undiscussed. For the current system, it is adequate, but it clearly omits certain crucial kinds of information, e.g., temporal constraints and formal I/O specifications.

Locality

In constructing a cliché library, cliché definitions should be as local as possible. Locality is defined in terms of how much of the cliché library needs to be considered when defining a new cliché. Ideally, when defining a new cliché, after having determined the most specific parents, only the definitions of these parents need to be considered. All other clichés in the library should be irrelevant to creating the new definition. This permits the modular expansion of the library with effort independent of the size of the library (or at worst dependent on the depth of the cliché hierarchy).

Clichés can never be fully local, however, since the clichés must communicate in some manner, e.g., through the use of common role names. It is the job of ıe Listener to combine cliché descriptions. To the extent that the Listener can successfully combine cliché descriptions, i.e., since all cliché combinations are not precompiled at the time of definition, the definition of clichés can be more local.

The desired locality property of cliché definitions is violated by the existence of certain gen^eric, overloaded concepts like inverse. The concept of inverse can be applied to a great many entities. Each kind of entity may have a subtly different notion of inverse, which prohibits using general purpose reasoning facilities to, for example, decide whether two objects are inverses. This creates a cross-product expense in creating the cliché library. Each time a cliché is defined, the list of generics must be consulted to see if information regarding the new cliché and the particular generic must be accounted for. This imposes a cross-product expense in the numbeı of generics and number of clichés.

Another sense in which locality is an important property is in the creation of canned text for a cliché. To the extent that the canned text is conditional on, e.g., the specific types of objects filling roles in the cliché, new clichés may need to be created to capture the novel conceptual combination and the documentation and semantics that go along with it.

Communication

There is a direct tradeoff between locality a cliché's ability to communicate with other clichés. One method of communication is via hierarchical abstraction. For example, a specialization of a tracking-system cliché may encode knowledge about repositories and may assume that the object it is tracking is a repository. In this way, the vocabulary and structure of repositories can be used to communicate, e.g., roles of the repository can be accessed and/or constrained. However, the tracking-system cliché does not need to know about all the specializations of repository. Thus, there is a tradeoff between the ability of clichés to communicate and locality.

Another way in which clichés communica^te is through generic concepts, such as inverse. In a domain model there may be a set of clichés that model things that exist in a context-free manner, e.g., books exist and they have authors, titles, and text; bicycles exist and they have a number of gears. These might be the environment clichés. There also may exist a set of purposes (needs), e.g., borrowing an object or using a vehicle for transportation. These concepts might communicate via the definition of cross-product clichés (while using hierarchical abstraction as much as possible) such as: book-as-an-object-to-be-borrowed or bicycle-used-as-transportation [43]. These clichés would define important aspects of the object for the particular purpose. In this way, creating a domain model will require a partial cross-product conceptual combination between interacting concepts.

Reuse

There is a relatively understated potential advantage to the reuse of data in software creation. If a data source is used in multiple projects, then further analysis in any of the projects may cause a rework of the data source thus propagating beneficial changes to all related systems. This has the effect of extending the normal testing paradigm ·· include multiple diverse sources of test data. This method has been used manually with success in at least one case. A bug was found in a section of space shuttle software [44] in which variables were not being properly reinitialized in a section of multiple-pass code. The instance that triggered this discovery was recognized as one of many multiple-pass code sections. The other similar code sections were checked and a number of analogous bugs were found and corrected. The effective use of cliché libraries in a semi-automated software development environment holds the promise for just this sort of added advantage in the creation of reliable systems.

Independent of the above debugging observation, reuse is generally considered to be a time saving technique. Effective reuse depends on the ability to access the appropriate units of knowledge. What terminology does a speaker need to know in order to interact effectively with the Listener? How is this terminology defined?

Two separate kinds of knowledge are captured with the definition of a cliché. First is the existence of some reusable concept as defined by the cliché definition. Second is a name for that concept. In the Listener system, a simplification has been made in which the name of the cliché-frame is also the name of the cliché. However, it should be possible in principle to separate these two, e.g., to implement a Listener using Spanish without changing the concept definitions. To handle this problem, a lexical pre and post-processor ought be implemented for the Listener system. Having done this, however, a host of potential issues is opened up relating to natural language understanding, e.g., resolving use of synonyms to an appropriate lexical entry. Some of these issues fall under the rubric of the disambiguation algorithms implemented by the Listener, others involve the looseness of natural language communication. The Listener system sacrifices the notion of general natural language interaction by assuming that the cliché library defines a technical vocabulary which the speaker will become acquainted with and which will be accepted as the standard for communication.

Given this assumption regarding the technical vocabulary, the question still remains: What terms does the speaker need to know?

The names of the clichés that you might expect the speaker to know are those in the "middle band" of the cliché library. The "top band" defines the ontology of the library and is so general as to be assumed in the communication. Use of top band terminology would generally be highly redundant. The lower bands of the library define very specific ideas, that, if given names, might have the following sort of cumbersome, long hyphenated names, e.g., tracking-system-with-history-and-operator-authorization-but-no-reporting, The speaker cannot be expected to know these names. Thus, the first thing the speaker needs to know are the names of the middle band clichés (e.g., tracking-system) and the names of relevant roles in these clichés.

Middle band clichés are refined to the lower levels along "axes of description." Each refinement embodies a decision about some aspect or parameter of the cliché. For example, in Section 3.2.2 on page 99 the parent cliché comparison-system is refined by clichés that exhibit a varying degree of influence on the objects being compared (monitoring-system, tracking-system, and control-system). This degree of influence is the axis of refinement. The refinement operators are another set of terminology with which the speaker needs to t familiar. These refinement operators should be applicable to many clichés as they represent concepts that are naturally mixed-in to other concepts. (These axes are currently not represented explicitly in the cliché library. Rather, they are used as an organizational aid in creating the library.) Wertheimer [45] reports on work to identify these sorts of reusable, mixin concepts in the domain of the design of rule-base inference engines.

Additional terminology includes the set of generic functions, e.g., inverse and compares, and relations defined at the top-level of the cliché library, e.g., tracks.

Not surprisingly, the speaker must be familiar with a significant amount of terminology, though, when compared to the total size of the library this knowledge may indeed be small. Nevertheless, as discussed in the introduction, a cliché library browser would be useful to assist the speaker in accessing the proper terms.

5.2 Using Clichés as Shared Knowledge

Since clichés are being used to capture shared knowledge, they play a critical role in determining the meaning of input to the Listener. A key philosophical problem with attributing meaning to a representation is that of meaning holism [46]: something's meaning is defined by the entire network it participates in. Along with this problem is the desire [47] to come up with a precise set of conditions under which it makes sense to say "X means <X>" (i.e., the statement of facts X is true iff the set of conditions <X> is true in the world). The motivation is that meanings must be precisely defined otherwise people could not communicate through an exchange of meaningful sentences. By these criteria, it is difficult for a computer program to capture meaning. Further, these are unnecessarily stringent conditions because they require super-human performance. People communicate using approximately shared concepts all the time.

The Listener's representation of an idea captures the idea as it is conveyed in a conversation. It does not capture some absolute definition of the idea, i.e., its true and precise "meaning." A conversation has the property that it leaves room for discrepancy of meaning. The acquired representation is good if the conversation succeeds. The conversation succeeds if, for example, the Listener can answer questions or fill in unstated details about the topic of conversation. In this light, the internal representation should be thought of as outlining a structure that constrains (but does not uniquely determine) the idea. The use of clichés in creating this representation has some important properties listed below.

The important observation, however, is that after a conversation of some detail is successfully understood, it is reasonably likely that the two parties involved have the same idea in mind. They may disagree on further extensions, but if they are in fact talking about different ideas then they have two representations that are largely isomorphic and for conversational purposes equivalent.

- Ideas can be described in a non-holistic manner. The current understanding of an idea can be captured by following all links to those concepts immediately connected to the idea. This is a *narrow content* view [48, 49] of meaning. It allows for each idea to be represented by relationships to a constellation of concepts each of which may change over ti[illegible]. As long as the relationships between the the idea and the concepts don't change, the idea's definition remains the same. This provides a way to move forward in a description without getting bogged down in definitions and natural-kind problems. As a result of this local definition, concept names inevitably carry some operational meaning.

- If all concept names are encoded to remove their associated English meanings, there is still a remaining structure which at least partially defines the concept. This structure is, however, unlikely to uniquely define the concept. For example, consider the following description of what looks like a stack:

  - stores(stack, elements), *(i.e., a true relation about stacks)*
  - size(in: stack, out: number), *(i.e., a signature with input and output designations)*
  - push(in: element, stack, out: new-stack) : size(new-stack) = size (stack) + 1, *(i.e., a signature with a constraint)*

- pop(in: stack, out: element, new-stack) :
  size(new-stack) = size(stack) - 1 iff not(empty(stack))
- top(in: stack, out: element)
- empty(in: stack, out: empty?) : empty? iff size(stack) = 0

The structure of the description could equally well apply to a queue since the axioms which define the LIFO behavior of the stack are missing. While it is true that precisely defined mathematical concepts (such as a stack) can be defined completely, such precision is not possible for the range of concepts used in a general conversation. Thus, the choice of names is important to the representation of ideas because it augments the structure of the representation by accessing meaningful primitive terms in the reader's mind.

The structure is useful in that there are certain syntactic rules which govern its well-formedness and can be used to detect problems in the description that formed it. Nevertheless, this points to the fact that validation will still need to be performed on any idea acquired using this kind of technology.

- The cliché library contains chunks of structure. The statements of the description access these chunks, explicitly and implicitly, and put them together. The representation of ideas is generally built up chunk by chunk, rather than link by link. Because of this, one can only build members of a small subspace of possible, syntactically valid representations. Furthermore, once a coherent description is achieved, adding new chunks is more and more likely to cause problems (unless the changes are orthogonal to the current state), since the new piece must fit the current structure.

5.3 Informality

The set of informalities that the Listener handles is based on a set of informalities that had previously been empirically identified as problematic [12] in requirements analysis.

A class of informality that is not handled is the more syntactic kinds of problems that might best be handled in the context of a more general natural language understanding system. Through use of the command language, the Listener assumes the speaker can create syntactically well-formed statements, e.g., without missing arguments or operators in propositional assertions (though roles can always be left unfilled in clichés), without extra arguments, or interchanged arguments. The Listener maps the speaker's statement directly into the knowledge-base and thus it must assume the basic syntactic correctness of these statements. An alternate design, that could accommodate more errors in the input, would involve adding one level of indirection, a translation step, between what the speaker says and what is asserted in the knowledge-base. In this way, for example, if the speaker reversed the order of some arguments, the translation step could propose a translation which reverses the order (to the correct order), presumably after trying the original order and detecting a problem.

It would be useful to be able to say that the Listener handled a "complete" set of informalities for some suitably interesting definition of complete. Such a definition could be presented based on the kinds of errors that can be made given the syntax of the underlying representation (e.g., in a propositional representation you can omit arguments, include extra or wrong arguments, and include extra or omit entire propositions). This was done in Section 3.1.2 on page 83 where the corpus of facts deducible by the speaker was compared to that deducible by the Listener. Still, such a characterization is merely one of symptoms and not of causes. Some causes of informality which the Listener does not handle are described below.

Implicitly required coercion is a basic form of informality. For example, if a role is filled with an object and this causes a type conflict between the role and the filler, it may be the case that the appropriate resolution is to coerce the filler to be of an appropriate type. In fact, any type errors (usually manifested as disjoint-type contradictions) are candidates for resolution by type coercion.

An alternative to coercion of the value is selection of an alternate role to fill. The Listener implements a heuristic which searches for other possible values to use in filling a role. Another plausible resolution would simply be to fill a different role, perhaps a sibling role (i.e., `a.b'` versus `a.b`) or a more deeply nested role (i.e., `a.b.c` versus `a.b`).

It may be possible to identify in a principled manner all the symptoms of informality. However, there are more causes than symptoms. An exploration of different domains of acquisition and analysis might lead to closure of the set of possible causes (or at least an asymptotic decline in the number of new causes discovered).

Order Independence

Order independence is important in the Listener system so that the speaker may have confidence while using it. The Listener is taking the role of an efficient automated assistant and is expected to produce consistent output given the same information content on input. This leaves the speaker free from worrying about unimportant details of their conversation.

There are stronger and weaker definitions of order independence depending on how the input and output are defined, e.g., is incremental feedback included in the output or just the description knowledge-base (DKB)? Does the input include just the statements made or the speaker's response to questions asked by the Listener?

Here, order independence refers to the contents of the DKB and the set of contentful statements made by the speaker, i.e., disregarding variations due to different orders of detecting and correcting problems.

The price paid for order independence is that of keeping around a large number of dependencies and of monitoring choices for new options. This becomes increasingly expensive over long periods of time. Since people are influenced by order of presentation, relaxing order independence would more accurately model human performance. Strategies for forgetting some of the support for well-accepted propositions could work to improve the overall performance of the system.

Incompleteness

The information in cliché definitions is generally declarative constraints. Each keyword is translated to have the effect of asserting the appropriate propositions when a cliché is instantiated. Some information in cliché definitions is computational data for procedures. For example, a cliché may have a declaration of `:equality-views` which specify: a set of roles that define equivalence between two instances of the type, and that it is particularly worthwhile to check for equality between instances of this type. The procedural component of processing for an equality-view is "for each new instance of a type check if it is equal to any of the old instances leaving appropriate dependencies behind which will detect the equality if it occurs in the future."

Incompleteness processing is an area where information needs to be specified in a more procedural fashion and therefore is only weakly supported by the current cliché definitions. Incompleteness processing is a qualitatively different problem than resolving other informalities since it requires potentially infinite speculation in a space of alternatives. The procedural component involves a search for components in one space that have no corresponding components in another space. However, the definitions of these spaces are not easily provided as declarative parameters to some procedure and thus need to be defined in a more involved procedural fashion.

Informality is a powerful communication mechanism. Seriously addressing informality provides ways to deal with imprecise communication, can speed up communication, and can overcome some of the brittleness in current knowledge-based systems.

5.4 Interface to the Design Stage

As shown in figure 5-1, the RA addresses one aspect of constructing software systems. Other work in the Programmer's Apprentice project [50] addresses problems closer to the implementation side of the process. The gap that remains is the process of high-level and low-level design of the software.

High-level design involves:

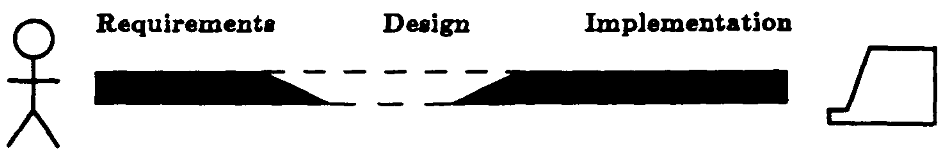

Figure 5-1: A Software Construction Process Model

- modeling decisions and representation decisions - the requirements acquisition will result in the identification of various objects that are part of the running system. For example, in the library problem there are books, users, and the library. The implemented system may have to model these objects internally and decisions about how to do this must be made. For example, imagine a library database system that tracked books for multiple libraries. For each library, the system could maintain a separate representation of the books in that library or the system could maintain a representation of all books in all the libraries and annotate each book entry with a description of the libraries it is in. As implementation is contemplated, many such decisions must be made.

- allocation of resources - this includes standard store versus recompute tradeoffs. More importantly, in a system that will be implemented with multiple pieces of hardware, decisions must be made about what loads will be placed on the various devices.

- allocation of responsibility for functionality - it must be decided which agents in the system and environment will be responsible for performing different functions. As the software is broken-up into modules, decisions must be made as to which module is responsible for a particular functionality or invariant.

- architectural decisions - the break down of the system into modules must be decided. Then allocation decisions, such as those mentioned above, must be made. The complexity of the architecture increases with issues of concurrency, distributed processing, reliability, and security.

The end product of the RA provides a high level specification of the functionality of the system. At this level, many of the terms used in the descriptions are still partially uninterpreted primitives, i.e., all their salient properties are not yet known. For example, in performing design it is often necessary to know which properties of an object are changeable and how, or which are directly observable and which must be derived. In the library example, the information that the text of a book never changes might become useful in designing a system that provided online access to information about the contents of the library.

The process of design involves many concerns which are not appropriate to consider at the requirements level. The data provided by the RA provides a context in which design decisions can be made. Further, the process of design is likely to elicit further descriptions of the environment and additional requirements that should be included in the RA's analysis. Thus, a design assistant in addition to requesting data from the RA would also augment the RA's knowledge about the evolving system description. In this way, the process of software development, from the Programmer's Apprentice automated assistant point of view (and other points of view [51]), is seen to be a continually incremental endeavor as opposed to the standard serial, or spiral, waterfall model.

Chapter 6

Related Work

This chapter consists of three sections of related work divided into the topics of knowledge acquisition. software engineering, and classification. The basic task performed by the Listener is one of knowledge acquisition. The RA performs a specialized acquisition task in the field of software engineering, namely requirements acquisition. Classification is an important underlying technique used by the Listener.

6.1 Knowledge Acquisition

This section begins with a framework for understanding research in knowledge acquisition. After presenting this framework, it discusses specific acquisitions systems. These systems are grouped according to their handling of specific issues relevant to knowledge acquisition.

Knowledge acquisition is a huge field using many different research strategies and problem solving methods. Acquisition methodologies and systems can be classified by a number of parameters:

- is the acquisition manual or automated
- is a domain model used (or not)
- is a problem-solving model used, i.e., does the system produce data for a particular performance program
- what kind of input data is used, e.g., natural language, structured text, graphical input, or numerical correlation ratings
- what is the quality of the input data, e.g., correct (precise), noisy or informal
- what acquisition techniques are used, computer-based techniques include [52]:
    - computer supported interviews

  - textual analysis
  - template filling
  - modeling in a particular language with a defined syntax and semantics (This often permits the use of simulation, consistency checks, testing and debugging.)
  - learning, e.g., analogy, explanation-based, or example-based

The acquisition systems discussed in this section span a range of positions on these parameters. They are presented, even when they take a radically different position than the Listener, to demonstrate the range of approaches and to suggest techniques that could be incorporated into the Listener. Another important parameter is the view of the acquisition cycle taken by a system.

The process of knowledge acquisition can be broken down into the following stages (as can requirements analysis):

- Elicitation - interview experts and get the raw data.
- Formalization - integrate the data into an underlying framework, correcting imprecision and some errors in the elicited data.
- Validation - integrate the data into a performance program and confirm that it produces correct results. The "performance program" could be a knowledge-based system or a person acting on the basis of the acquired knowledge.

A number of knowledge acquisition systems assume that the experts from whom the knowledge is elicited have the correct knowledge readily available. If an expert can perform a task, it is assumed that he must have a correct, formalized version of the knowledge available. These systems divide the acquisition problem into a two phase process of elicitation and validation (and may leave validation to another tool). Such systems ignore the formalization stage that the Listener focuses on. Two examples are discussed below.

KRITON [53, 54] uses an elicitation and representation cycle. It is a domain and problem independent elicitation tool. It assumes the rules, concepts, and attributes it elicits are correct. These definitions are delivered to a representation medium where they are operationalized and debugged. Debugged, in this case, means the acquisition of strategic and control knowledge that guides the application of the acquired knowledge. KRITON's acquisition cycle is being revised to include a model of the problem solving process so that strategic knowledge can also be acquired during elicitation. Nevertheless, the only sense of formalization is the translation into the representation language. No provision is made for the acquired data simply being wrong.

An example of the strategic knowledge acquired during representation is as follows. Suppose in a system for classifying mushrooms, the knowledge that poisonous mushrooms taste bitter has been acquired. Using this knowledge, i.e., tasting the mushroom, may not be a very good idea and should generally be avoided. Control knowledge can be acquired that limits the use of this knowledge to appropriate circumstances. No provision is made, however, for the fact that this knowledge may be wrong, i.e., that poisonous mushrooms may not actually be bitter. There are two ways that might be used to detect incorrect knowledge. First is to use the knowledge and trace a wrong conclusion to the specific piece of knowledge. Many other systems do just this. Second is to use knowledge of the domain ahead of time to conjecture that the knowledge may be wrong, as the Listener might. This requires the use of domain models which will be discussed below.

Another example of an elicitation-validation system is ASTEK [55], which is concerned with acquiring data using multiple, editable presentations based on a consistent underlying representational framework. These multiple presentations assist in acquisition by providing clear external data forms and convenient input formalisms (e.g., object reference by pointing or description, versus naming). Still, it is assumed that the knowledge provided is correct, correct by construction. In essence, ASTEK assumes that by assuring the correct syntax of the knowledge, entered using the most natural available presentation, that the semantics will be correct.

The Listener shares many traits with other knowledge acquisition projects. It also differs in a number of ways, particularly in the combination of issues it addresses. The Listener system is: automated, knowledge intensive (using domain models), formalization centered, integrative, capable of accepting noisy and informal input, unintrusive (user controls the interaction), instance (versus type) centered, and reuse centered.

Many knowledge acquisition systems learn by performing some form of "curve fitting," i.e., the acquisition of general theories from data points. The implicit assumption underlying these systems is that the data is noise and error free. Another manner of acquisition, that admits the possibility of noisy data, is by debugging, i.e., take the theory as input and debug it. Debugging can proceed from at least two kinds of data: input-output data or domain knowledge. A theory can be tested to see if accounts for the data. If it cannot, then it must be repaired. Individual data points, however, provide no rational basis for generating repairs. Domain knowledge, on the other hand, characterizes the kind of expected behavior without using specific examples. Using the curve fitting analogy (though not to push it too far), domain models provide information about the *distribution* of data. Distribution information provides data about typicality and can be used to debug, repair and elaborate a theory. If appropriate, atypical components of the theory can be accommodated by confirmation from the expert. The Listener takes the domain modeling approach.

To see the difference in the two approaches, consider the following example. Say that the MIT id numbers of foreign students were fed as data to a knowledge acquisition system. (Foreign students tend to be assigned id numbers beginning with 888, other students typically are assigned their social security number.) Generalizing from the data might cause an acquisition system to deduce that all foreigners are issued social security numbers starting with 888. Given the proper domain knowledge (regarding the distribution of social security numbers, that the prefix denotes geographic region, and how MIT typically assigns id numbers), this theory could be challenged even though all the data supported it.

Another way to view this issue is as the acquisition of instances versus types. The goal of an acquisition system is often (though not always) to acquire a more general type description by abstracting or generalizing from the input data (instances). The individual data points are uninteresting (and assumed to be correct and well-formed). In such cases, instance data tends to be accessible and observable which is another reason it is uninteresting. The Listener, however, tries to acquire a detailed description of an instance of a descriptive theory. This theory is not observable and potentially needs to be debugged (e.g., a software requirement). The Listener attempts to understand the details of the description in the context set up by the domain model rather than generalizing it and extracting reusable knowledge. Other systems do try to acquire data to fill out a previously acquired conceptual framework. In such cases, however, the acquisition usually amounts to filling in slot values.

The Listener system contains two domain models: the cliché library and the evolving description. Knowledge acquisition techniques are relevant to the acquisition of both the Listener's domain models. The use of some knowledge acquisition method is presupposed to produce the Listener's cliché library. The cliché library is a corporate or community resource (e.g., as CYC [56] is intended to be) that contains the "common-knowledge" of the domain. The Listener integrates the new description specific knowledge with the cliché library as it acquires a coherent description. Recognition of the importance of codifying domain knowledge is a theme that the Listener shares with systems such as Φ-NIX [57] and DRACO [58].

BLIP

BLIP [59, 60] is, perhaps, the knowledge acquisition project closest in goals and spirit to the Listener, but of a more limited scope in terms of modeling. BLIP assists a user in a process called *sloppy modeling.* Sloppy modeling means the user is not required to develop a complete and well-structured model beforehand and then transfer it to the machine. The system organizes and completes the knowledge as it is entered by the user, making implicit data explicit. BLIP also provides feedback on what it is doing, to influence user's activities. The system does not assume the user is an oracle of knowledge. BLIP and the user act as partners that cooperate and interact.

The goals of the BLIP system, which are quite compatible with the Listener's goals, are:

- flexibility - supporting an unintrusive acquisition that lets the user say things as they occur to him, using information when it becomes available and not requiring extensive interviewing.
- reversability - support for non monotonic acquisition.
- integrity and consistency maintenance - integrity refers to consistency between multiple data sources.
- liveness - a live interface, i.e., displaying changes to the model to permit the user to see its evolution.
- inspectability - e.g., providing browsers to view the acquired data.
- transitionality - e.g., switching to a mix initiative dialog when required to wrap up loose end.

BLIP uses a logic-oriented representation similar to CAKE's, a sorted logic with facts, predicates, and rules. Predicate definitions state the sorts of their arguments. Facts are stated using the predicates. All propositional statement input must be well-formed according to the syntax of logic.

BLIP's interface is through three windows that display the most recently accessed acquisition model. These models include a presentation of: facts, predicates, sorts, rules, metapredicates, metafacts, and agenda items. Metapredicates are templates for rules . Metafacts are instances of metapredicates that generate rules. BLIP provides immediate feedback of changes in the models as they are deduced. Deductions are simple structural conclusions based on the syntax of the logic. BLIP also has an agenda of open ends (pending issues, integrity violations). Agenda items can be automatically addressed, e.g., predicate definitions and the sort hierarchy can be automatically generated.

BLIP's primary limitation is that it does not make use of a domain model as a source of knowledge and expectations. It is more like a logical specification notepad (a subset of the described behavior of the Listener). It uses simple correctness rules, e.g., any fact must use defined predicates (which must use defined types in the sort hierarchy) and rules imply predicates and facts. It also makes a simplifying assumption that no term can be a member of more than one sort (except for super/subsorts).

(As a system, BLIP is ambitious in that it supplements its manual modeling techniques with an automatic machine learning component. It manages the integration of manually acquired data and automatically induced data. BLIP debugs the rules it forms as exceptions to them are found. When enough exceptions are found, a conceptual realignment is performed and new rules, using the new concepts, are formed.)

Issues in Knowledge Acquisition Systems

The remainder of the description of related knowledge acquisition systems divides the systems by their positions on particularly important issues. Most systems can be evaluated with respect to many of these issues. Each system is discussed under the issue which seems most central to its operation or to which it provides the most relevant example.

Many acquisition systems assume that the data they acquire will be used as input to some performance program, typically an expert system. An important issue is whether or not an acquisition system makes such an assumption and, if it does, how it makes use of the assumed problem-solving model.

It is not clear that the Listener assumes a significant performance program model. The descriptions it acquires are specifications of valid behavior or existence. The performance model is basically that the artifact described (using informal description techniques) can exist and operate in the world. The knowledge acquired is not limited by a particular syntax or semantics, except for that of the chosen underlying logical representation. An operational semantics to provide a source of expectations is missing (but in the requirements analysis domain would be an import extension). For this reason, the use of a domain model is crucial to the Listener's capabilities. The domain model provides the expectations that would otherwise be provided in a weaker form by the problem-solving model.

A companion issue to the assumption of the problem-solving model is whether or not the system acquires strategic knowledge used to control the use of the other factual knowledge it has acquired. Two of the other issues also concern the focus of the acquisition process. Some systems focus on acquiring a problem vocabulary, i.e., simply on establishing a conceptual model that defines the domain of discourse. Other systems focus on fine tuning there strategic knowledge to produce optimal behavior on a test suite.

The final issue regards the nature of the input data. Some systems attempt acquisition from natural language input. This complicates their task greatly.

The Listener's overall position on these issues contrasts greatly with the typical system. The typical acquisition system assumes a problem-solving model, i.e., that the data acquired will be used to run a performance program, and further assumes the availability of an expert to provide correct, precise input data. The knowledge of the expert also tends to obviate the need for a domain model. In contrast, the data acquired by the Listener is simply a description. There are no experts who can provide the description precisely and thus the input is assumed to be noisy and informal. A domain model is needed to make up for both the lack of a problem-solving model (and the source of expectations it can provide) and the lack of expert data input.

Systems that Assume a Problem-Solving Model

Many acquisition systems make use of a model of the performance program which is to use the data acquired. This model provides a syntax and semantics for the language in which the acquired knowledge is expressed and thus provides a source of data to use in checking the acquired knowledge. The model can also be used to assist in querying for knowledge.

One of the earliest systems to assume a problem-solving model was TEIRESIAS [61, 62]. The model was that of an EMYCIN rule-based system. Knowledge of the rule structure permitted derivation of rule models which facilitated the comparison of new rules to old ones, allowing the old ones to provide some notion of typicality and correctness. In this case, the rule models provided domain specific debugging information. The problem-solving model also provided an independent source of operational debugging information. Similar rule-base maintenance procedures can be seen in the THX1138 [63] system for acquiring rules for diagnosing building structural problems. In this case, rule debugging is performed by hand after inductive acquisition (by EVITA) and then all rules are automatically verified (by CIRCE, a system similar to OPMAN [64]) using the static semantics of the rule-based programming language.

A more recent project that relies heavily on a problem-solving model is CGEN. CGEN acquires knowledge to be used by the MICON Synthesizer Version 1 (M1) rule-based system for the design of small computers. CGEN assumes the existence of a hardware engineer that has the needed design knowledge. One goal of the system is to eliminate the need for a knowledge engineer by interfacing directly with the expert using familiar interfaces, e.g., schematics. When M1 cannot find some design information it needs (e.g., a template to match a particular design to the use of a particular component), it produces a design-state file which captures the current problem and the state of the solution. This file and information provided by the expert as to the missing solution is used by CGEN to create an appropriate rule to update the knowledge-base.

The rules generated use data patterns that are generalizations of the current exact design state. Design state parameters are appropriately generalized, based on information provided by the domain expert, to make the rule more applicable. CGEN does not perform explanation-based generalization which would require an extensive and deep domain model. CGEN's capabilities revolve around its understanding of the performance program's problem-solving method.

Both M1 and CGEN assume the input provided them is correct. M1's task is to design the specified hardware, not to debug the specification. CGEN creates a rule applicable to a particular problem and its generalizations, but it has no way of judging whether its solution violates higher level design principles.

Other examples include the following systems. MOLE [65] acquires knowledge for a heuristic classification problem solver that uses a restricted problem-solving paradigm involving the observation of symptoms and finding a hypothesis that covers all the symptoms. ROGET [66] is another system for building EMYCIN rule-bases for various kinds of different problem-solving models. ROGET uses knowledge of previously acquired systems to guide it in asking questions and generating examples. Descriptions of desired input information are combined with previous domain examples and used to prod the expert into providing the correct information (by analogy). However, the information provided is not debugged using knowledge from the other domains. UNDERSTAND [67] is a system for constructing solutions to state transition problems described with a set of written instructions. In this case, the problem solving model permits deriving operator definitions from the input description. GPS is used to find a combination of operators that permits achieving the specified goal.

Acquiring Strategic and Control Knowledge

Strategic knowledge is used by an agent to decide what to do next. However, experts cannot provide this knowledge very readily or accurately. Therefore, systems that acquire this sort of knowledge should provide good examples of the formalization process. Many systems exist that focus on the acquisition of strategic and control knowledge, though often this knowledge is treated as a problem to be solved during implementation of the system that uses the other knowledge that has been acquired.

ASK [68] is an assistant that engages in a learning dialog and acquires strategic knowledge by generalization and syntactic induction. ASK assumes that the substantive knowledge (i.e., the non-control) knowledge in the performance system is correct. ASK also assumes a problem-solving model which provides a representation for the strategic knowledge. The model of control consists of a pruning and a conflict resolution stage. ASK engages the expert in an elicitation dialog during a run of the performance system and the expert is asked to express his ordering preferences on actions and justify these preferences. During this process, new domain knowledge may also be acquired in the form of new domain attributes, e.g., time-required or cost.

ASK represents the acquired control knowledge in the form of meta-selection rules. The knowledge is acquired based on instances of specific behavior in a running performance program. ASK's job is to create theories that will account for behavior in the particular training cases run. The expert's job is to provide the correct control decisions in the particular situations. The underlying assumption is that these decisions transfer to other cases, i.e., that the knowledge provided is correct and does not have to be formalized. An alternative approach would have been to elicit control rules, initially in absence of particular examples, and debug these rules with both domain knowledge of how decisions are actually made and specific test cases. In this way, the acquired knowledge would be less influenced by the specific choice of test case data.

Experts can readily specify design parameters and constraints. They have trouble describing combinations of design decisions or specifying an order in which to address them. During elicitation of such information, experts have trouble sticking with their identification of subtasks and their specification of ordering decisions. SALT [69] elicits design considerations and interprets the individually acquired pieces to represent their relationships. It starts acquiring knowledge from the bottom (details) up. SALT builds strategic knowledge from its input data which falls into the categories of: parameter determination procedures (formulas), constraints, or fixes for constraints violations. It assumes a propose-and-revise problem-solving strategy. It groups primitive design steps in subtasks that minimize dependencies and the chance of backtracking. Each subtask is defined as a local-constraint satisfaction problem. This technique of minimizing dependencies uses knowledge of the problem-solving model without relying on specific test cases to drive the acquisition.

Systems that Acquire a Problem Vocabulary

Some acquisition systems address, as their central problem, acquiring a domain of discourse, a vocabulary to use in problem solving. This can be especially difficult when multiple experts provide data and use conflicting terminology [70]. Equally difficult can be the problem of incorporating new information into an existing knowledge-base. The terminology used by the current expert may disagree with the terminology used in the knowledge-base. (Note that solutions to this sort of problem give leverage on the cliché access problem, since the problem addressed is *finding* where a new term fits into an existing knowledge-base.) KNAC [71] addresses this problem. The basic technique used by KNAC is a structural similarity matcher. KNAC also provides for user verification of acquired information.

OPAL [72] is a knowledge editor for a cancer therapy management system called ONCOCIN. It also has an internal model of the problem solving process and a domain specific task model (a conceptual model of domain specific concepts, e.g., chemotherapy, drug, toxic reaction, and laboratory test). It uses the domain specific model to guide its foı based (fill in the blanks) acquisition. One of its major goals is to acquire a vocabulary of treatments, tests, and other facts for use in ONCOCIN, i.e., facts needed to fill out the conceptual structure of the knowledge-base with data. It does, however, assume that an expert presents only correct facts. (As with other systems that use a problem-solving model, it can translate the data acquired into an intermediate representation and then to a target formalism, e.g., ONCOCIN rules.)

Other examples included the following systems. STUDENT [73] performs acquisition in the domain of data and statistical analysis. It provides a builtin conceptual framework to guide the acquisition of specific facts. Ontological ana is [74] is a methodology, versus a system, for preliminary analysis of a problem solving domain. Part of the information it helps acquire is a static ontology, i.e., what exists in the problem domain. An ontological structure specification language called SUPE-SPOONS is used. SUPE-SPOONS statements are either domain equations (type definitions) or domain functions declarations (signatures). Analysis of these statements, including the use of generalization and aggregation, permits establishing a domain vocabulary. Type and units checking can be performed on the domain equations to ensure consistency. Even though no domain models are used, this is another example of how the representation language can provide a source of data integrity constraints.

Systems that Twiddle Weights

Some systems acquire data regarding the weights of various belief or certainty measures in an effort to optimize their performance on a suite of test cases. This sort of induction is somewhat puzzling in that it ignores rational explanation and causality and simply performs a form of curve fitting. Thus it requires, the utmost confidence in the structure, coverage, and content of the knowledge base. Most of these systems also have other acquisition facilities.

AQUINAS [75] is a workbench that interviews experts and helps them: organize, analyze, test, and refine their knowledge bases. Data is elicited and represented in a *repertory grid*. The columns of a grid are solutions (elements) and the rows are traits. Grid entries represent correlation measures between solutions and traits. Knowledge is tested by running consultations. Refinement involves changing weights (i.e., grid entries) to make the acquired network have the highest average correlation score over the test suite compared to expert opinion. (Among the other features of this tool is a grid analysis which can detect similar solutions or traits and try to either eliminate the redundancy or cause the expert to make a useful distinction. Grid analysis can also look for correlations in trait occurrences.)

Other relevant systems include one for acquiring the importance of concepts in newspaper articles [76]. This system forms a network representation of concept connectivity and runs an activation spreading algorithm over the network to come up with concept importance values. These weights characterize the test data presented. It is interesting that they do not suggest using this knowledge to debug, evaluate, or critique an unseen article. SEEK [77] analyzes rule set performance on a test suite to suggest rule changes. Modified rules (generalized or specialized rules, i.e., changing the number of features required for a rule match) are tested by seeing how they effect performance of the system on the test suite.

Natural Language Acquisition

Some systems pay relatively serious attention to acquisition from natural language input. This can add complexity to a system without contributing much in the way of additional knowledge. Natural language is compelling as an input medium in that it allows a speaker to make use of many efficient and powerful communication techniques. This often shifts the burden onto the hearer (Listener) to properly interpret the utterance. One powerful technique in natural language is ease of reference. ASTEK [55], described above, avoids natural language input and provides a nice solution to the reference problem by permitting pointing (to an object on the screen) or descriptive reference versus pronoun or naming reference. The Listener provides some of the most important benefits of natural language input by supporting informality in the input data, but it does not have to address other difficult natural language problems such as parsing.

KBAM [78] acquires knowledge from natural language explanations. Explanations refer to both types and instances. The type structure is acquired and the instances are checked to see that they obey any type restrictions. This acquired domain model is relatively simple. Utterances in the explanation are divided into nine types including: definitional, entity-tagging, causal, episode-entity and goal-specifier. Each utterance type has an intention and thus a potentially different acquisition routine to explain and check it. Since KBAM takes natural language acquisition seriously it has a lot of overhead associated with parsing the input and, for example, representing domain-independent consensual world knowledge like what objects are entities and what are episodes. The natural language emphasis seems to indicate that most problems discovered will be category error kind of problems.

As a side note, KBAM uses SUDs (semantic update demons) to achieve order independence in the construction of the knowledge-base. Like the Listener, the system is concerned with evolution. SUDs appear, however, to only account for monotonically increasing information. In any case, the recognition of the evolution problem is important.

KLAUS [79] is a system that acquires a domain model by being told about it. It uses natural language input and learns about natural language at the same time. It starts with some seed concepts such as: person, physical-object, and measure. It is a mixed initiative system that provides immediate paraphrase feedback of user input. Its principles of knowledge organization are standard, e.g., there are things, there are subclasses, and there are relations. The distracting thing about this system is that thrown in with everything else, it asks questions to help it acquire natural language information, e.g., about verb tense forms. This lexicon acquisition is useful in that meaning can be inferred from, e.g., a verbs passivity, transitivity, or reflexiveness. However, acquisition of such lexical knowledge seems like a problem better handled in other contexts, e.g., by automatic analysis of a large corpus of sentences [80].

6.2 Software Engineering

Work on the RA fits in between two larger bodies of software engineering work. On the one hand, there is work concerned with the derivation of a specification from an initial formal description and the validation of that specification. The key difference between these kinds of projects and the RA is their assumption of the existence of a formal description from which to begin the process. On the other hand, there is work concerned with the elicitation and syntactic representation of informal requirements (or specifications). This work does not make use of a formal representation and thus the description that is acquired is not very useful for computer-based analysis. The RA bridges the gap between these two ends of the spectrum. The heart of the RA is a formalization process that captures and expands informal input into an initial formal representation.

A comprehensive framework for the development of a Knowledge-Based Software Assistant (KBSA) is outlined in [81]. The KBSA project proposes to provide automated assistance for all stages of software development. This requires the formalization and machine capture of all artifacts and decisions made in the construction of a piece of software, including requirements and specifications. Knowledge-based reasoning can be applied to the computer-based representation of the software and assist in making decisions. The KBSA project proposes that most software development activities will occur at the requirements and specification levels, with the automatic derivation (and rederivation) of an implementation. The project is divided into *facets*, one of which is a requirements facet. This facet will be discussed in its appropriate place below (as will the specification facet). The KBSA report outlines a new paradigm for software construction in which capabilities such as those demonstrated by the RA will be crucial.

Formalization Tools

This section describes research that tries to bridge the gap between informal and formal representations. As Balzer [1] points out: "informality will always exist during the formulation of a specification. The issue is whether the informal form is explicitly entered into the computer ... or whether it exists only outside the computer system in someone's head or written somewhere in unanalyzable form."

SAFE

The SAFE [1] project had goals closest to the RA. SAFE is a system that assists in the acquisition of an operational specification from informal natural language input. The input to the system is a parenthesized natural language specification (allowing the system to avoid syntactic parsing issues). The output is a runnable program written in a relational database language, AP3. The attention to informal input was motivated by the fact that most research at the time (but to a lesser extent today) focused on formalisms for program specification without concern for the difficulty of creating the initial formal specification. Thus SAFE and the RA differ from most current work on software requirements tools (e.g., [9, 8, 10, 82], described later) that focus on the validation of a requirement that is already stated in a formal representation.

SAFE was concerned with resolving informality relative to three different purposes:

understanding the structure of the domain, completing partial or ambiguous input phrases, and creating an operational program that meets the specification. The kinds of informality that SAFE corrects in its input include: missing operands, omitted actions, incomplete references, implicit type conversions, terminology changes, refinements or elaborations, and implicit sequencing decisions.

Many of SAFE's corrections remove informality due to the use of relatively unconstrained natural language, where the RA uses an expressive structured command language. For example, SAFE resolves pronoun references and other partial references such as: "the red one" and "the book." In general, SAFE focuses on completing partial descriptions using the context provided by either: the initial description, or by a simulation of the derived program, or by a limited domain model. It should also be noted that SAFE handles a number of informalities that the RA does not even address due to the fact that SAFE produces an operational program and must therefore resolve problems with the well-formedness of that program.

The RA focuses on the correction and elaboration of requirements descriptions. The RA relies more heavily on the context provided by domain knowledge. SAFE and the RA both address the crucial problem of problem acquisition from an informal input though they use somewhat complementary notions of informality.

RML

RML [23, 24] is one of the first applications of AI knowledge representation techniques to the representation of software requirements. RML is a formal, frame-based representation for requirements knowledge. Its basic ontology includes entities, activities, and assertions. Each ontological category has a rich semantics, e.g., an activity has the following parts: input, output, control, precondition, postcondition, activation, condition, stopcondition, and part. Time is also represented in the semantics of RML. Finally, RML supports a number of rich abstraction mechanisms, e.g., aggregation, generalization, specialization, and classification.

RML can be used to represent an informal description in a computer-based representation with formal semantics. For example, Greenspan [24] defines a mapping from SADT to RML that provides a formal semantics to the information in the SADT model. Before an informal description can be refined and elaborated, it must be represented in a machine manipulable form that preserves its intended meaning. RML provides this crucial prerequisite in acquiring problem descriptions. The RA's internal representation is influenced by the definition of RML. RML is a richer and more precisely defined language. The RA, however, pays more attention to analysis of the informal representation using clichés and hybrid reasoning techniques (neither of which are part of RML since it is basically a language definition).

KATE

The current research with goals most similar to the RA is the KATE [83] system. Fickas has proposed an interactive system that will provide assistance over the entire requirements acquisition process emphasizing specification design. The system will critique an evolving specification [84, 85] and will eventually be capable of generating examples to support its observations. To make this feasible, knowledge of a particular domain (resource management) will be provided to the system. The current emphasis in the project is to take a valid specification (syntactically correct, consistent, and unambiguous) and try to poke holes in it, i.e., critique it. This complements the RA's function which is to provide a syntactically correct, consistent, and unambiguous specification.

In emphasizing the notion of critiquing a software specification, Fickas has framed the problem as analyzing a specification in the context of a particular set of goals (needs in the RA's terminology). The priorities given for each goal form, in essence. the requirements of the system. (The project does not yet provide significant support for the acquisition of these requirements (priorities). Currently the policy weights are provided as input to the critiquing program.) Typical goals in resource management include: maintaining a large inventory (so people have a lot to choose from), giving patrons lots of resources, and maintaining patron privacy. A domain model is used to determine the specification's position with respect to the desired goals. This domain model includes usage scenarios (e.g., the case of a user removing a resource in the name of another user) that are positively or negatively linked to goals (e.g., the previous scenario is negatively linked to avoiding theft). An interactive matcher is used to align components of the specification with components of the domain model. Once this alignment is obtained, the specification can be criticized for compliance with the required goals.

KATE currently addresses an important form of requirements needs/goals analysis. Its representation of the problem seems to fit in well with Potts's issue-based representation (describe later, satisfying a goal is an issue). Both representations require some amount of manual intervention to make up for a lack of deep analysis, but both provide good ways to structure and capture the reasoning about a requirements analysis.

KBRA

The Knowledge-Based Requirements Assistant (KBRA) implements the requirements facet of the KBSA. The goal of the KBRA [86] project is to bring the computer into use at the very beginning of a software project. The KBRA accepts informal statements of requirements, from multiple viewpoints, and partially captures the semantics of these statements in an effort to incrementally formalize the requirement.

The metaphor of the KBRA is that it manages an engineer's notebook. The presentations it makes available to the engineer are an intelligent notepad (which allows the entry of free text and performs some simple lexicon parsing on it), a context diagram, a state transition diagram, and a calculator spreadsheet (where constraints can be defined and propagated). Like the RA, the KBRA can generate a requirements document (in 2167 style) that brings together information from all the different viewpoints and can be used to communicate the requirement. All presentations update and are updated from a central evolving system description, though communication and sharing between different presentations may be sparse.

Acquisition in the KBRA is in a "catch-as-catch-can" style emphasizing breadth of representation over formalization. A library of reusable requirements components is also included in the KBRA system. These may be used to include information in a cut and paste style. Reasoning processes in the KBRA include inheritance, classification through the use of special purpose decision tables attached to library components, and limited constraint propagation. The goals of the KBRA system are similar to those of the RA. The RA emphasizes depth of reasoning and reuse over breadth of input and representational ability.

WATSON

WATSON [87] is a system that acquires formal behavior specifications from informal scenarios, i.e., traces of typical operations. The input is restricted to these scenarios and the domain is restricted to telephone services. WATSON includes a model of the common-sense notions for the telephone domain that is used in debugging the specification.

The scenarios are represented as logic rules and WATSON debugs these rules, repairing contradictions, eliminating dead-end states, inferring missing rules, and checking the completeness of the stimulus-response pairings. WATSON is restricted to the episodic, finite state telephone domain. These restrictions allow WATSON to use more specific techniques, aimed at the syntax of its rule representation, to resolve informalities.

Formal Methods

This section describe methods that utilize a formal representation and well-formed formal input to support program derivation and validation. Their strength lies in the analyses that can be performed when dealing solely with formal data. The system models that are created can be validated by a variety of techniques including simulation, symbolic execution, theorem proving, and rapid prototyping. Eventually, the RA's RKB might provide the formal input required by such systems.

Central Repositories

A number of systems provide a central repository for structured requirements information. The central repository typically supports the construction of a requirement by multiple authors and the generation of informative comprehensive reports. The RA provides a central repository and can generate a summary document report, however, it does not support multiple (concurrent) authors.

The PSL/PSA [88] system supports a methodology for creating structured system documentation. The primitive constructors of the problem statement language (PSL) include: objects, properties, values, and relationships. Built on top of this is a system description ontology that includes the concepts of: system I/O, system structure, system size and volume, and many other characteristics. The user of PSL/PSA creates a requirements database from which reports can be generated. The problem statement analyzer (PSA) creates these reports which include: database modification reports, cross-reference reports, summary reports, and analysis reports, e.g., the data process interaction report (gaps in information flow or unused data objects).

The Software Requirements Engineering Methodology (SREM) [8] makes use of a control and data flow specification representation called an R-net. R-nets are used to create flow graphs of required system processing steps. A data flow analyzer is used to analyze the R-net flow graphs and detects incompleteness and inconsistency. An R-net simulation capability is also provided. SREM also supports an extendable version of a PSL/PSA-like system requirements ontology that allows the definition of new concepts, attributes, and relationships. A central database is used to store the system description. The user can specify the production of customized reports to assist in the analysis of the system description.

Formal Specification Languages

Formal specification languages have a precise semantics that can be used to support executability or proof procedures (or both). Systems that make use of these languages typically assume that the user can provide at least an initial high level specification. Writing such a specification is typically as error prone as writing a program. Provision for informal input is not made. Specification written in these languages can be checked for inconsistency and incompleteness. The RA extends these techniques to the domain of informal descriptions.

The Requirements Language Processor (RLP) uses a stimulus-response model to capture requirements for real-time systems. The stimulus-response model can be used to generate finite-state machine and petri-net representations. These representations support a number of analyses including the detection of incomplete transition specifications, inconsistent state transitions, and redundant specification of a known transition.

In [10], Kemmerer discusses a specification testing approach that operates on a state-machine based formal specification language called Ina Jo. The Ina Jo language includes constructs for describing types, constants, variables, definitions, transforms, initial-conditions and criteria. The criteria specify the goals the system is to meet and the transforms describe the valid state transitions (using first-order predicate calculus) from a set of initial conditions. As an example, a formal requirements model and specification of the library problem is presented. Symbolic execution is used to test whether the system specification description meets the requirements criteria. The specification can also be used to facilitate the construction of a rapid prototype.

Operationalization adds a powerful validation tool to a system. In order to be useful, however, behavior must be specified and modeled quite precisely. Requirements define a predicate on acceptable behavior without necessarily providing the necessary operational information. Certain design decisions (how-to decisions and modeling decisions) often need to be made before enough information is available in a requirement to support operationalization.

Paisley [89, 90] uses an operational, executable model for describing embedded systems. Like the RA, both the environment and the system are modeled. The unit of specification is the process. A process is defined by a set of possible states and a state successor function. Zave contrasts this to a data flow representation in which data flow can be confused to imply control flow. Processes provide a superior encapsulation. A Paisley specification is inherently precise and unambiguous. It is internally consistent by construction if it can be shown to be deadlock free. A specified system can be tested and debugged by running a simulation.

Larch [91] introduces an interesting twist and observation on formal specification modeling and methodology. Larch uses a two-tiered representation. A language interface component is used to connect Larch to an implementation language. Pre and post conditions are used to specify the required interfaces. The second tier is the Larch shared language (which is implementation language independent) in which specification traits are defined. Traits are defined using equational axioms. Traits may be composed to facilitate the construction of larger specifications. A handbook of reusable traits is provided to facilitate this construction. As with other languages, tools are provided to prove equational theories. These tools support semantic checking of user provided queries. Wing [92] makes an interesting observation regarding the use of Larch in the formalization of an informal requirement by hand; significant questions arise during the attempt to model the problem using the Larch language. The process of formal modeling forces the person doing the modeling to think more precisely about the artifact being described. This is exactly the the leverage the RA tries to achieve during its automated formalization of the input description.

An Issue-Based Framework

Potts proposes a method for keeping track of the reasoning behind decisions made in the construction of a requirement [93, 94] (or any other decision making process). While this is not strictly a tool for eliciting or analyzing requirements, it addresses the problems involved with managing changing requirements. One way to avoid problems with changing requirements is to anticipate changes, but this is difficult. Sensitivity to change can, however, be decreased by representing the reasons behind decisions, thus permitting identification of decisions and components effected by a change.

Potts defines an issue-based framework, which includes as primitives: artifacts, steps, issues, positions, and arguments. Arguments support (or object to) positions, which respond to issues. Issues may also imply information about positions and arguments. Resolving an issue may cause a step to be taken, which creates a new artifact.

A domain model might include some of the decision making information needed in a requirements acquisition. But in the absence of such domain knowledge or deep reasoning, this issue-based representation supports the capture of the analyst's own reasoning. The main missing component in this work is that it does not yet propose the detection of conflicting arguments or the analysis of arguments for their influence on the requirement.

The RA tracks decisions by setting up the proper dependency network, but its arguments are not represented in as great detail as the issue-based representation would support. Potts's work provides a much richer structure for representing decisions and would make a good addition to the RA. Its style of argumentation could be integrated into the RA as an additional source of informal input, i.e., position and argument statements would also be considered input to the RA and thus be analyzed using the RA's informality resolution methods.

Knowledge-Based Specification Assistant

The Knowledge-Based Specification Assistant [16, 95] implements the specification facet of the KBSA software assistant described above. The project is concerned with transforming an initial set of requirements to complete specifications. They assume that a requirement has a rich semantics (i.e., it has been formalized to a large degree) and that "specifications are simply those parts of requirements that have been formalized well enough to have an operational semantics." A major twist from most formal specification projects is that the specification assistant uses high-level editing commands that are not necessarily correctness preserving. These commands transform and augment a high-level, environment centered specification (problem specification) to a low-level, system centered specification (solution specification). The catalog of editing commands includes 70 high-level editing commands (non correctness preserving), e.g., adding a parameter and adding a precondition, and 20 meaning preserving transformations.

ARIES

ARIES [96] is a newly started project to integrate the specification and requirements facets of KBSA into a system for the Acquisition of Requirements and Incremental

Evolution of Specifications (ARIES). Its focus will be to treat the development of requirements and specifications as part of one continuous process, utilizing a single underlying representation. The system will support a gradual formalization of a wide spectrum of descriptions from highly informal to completely formal. Informal and formal descriptions will coexist inside of ARIES. This integration will attempt to support the breadth of acquisition capabilities provided by the KBRA and the depth of analysis and operationalization provided by the KBSA. The capabilities demonstrated by the RA address a task much like that required to bridge the gap between the KBRA and the KBSA. Work on the RA has influenced the parties involved in the ARIES project particularly with respect to the goal of gradual formalization of informal descriptions and the need for domain knowledge to support the process. The results of this effort should be exciting.

Informal Methods

This section describes methods that assist in the acquisition (elicitation) of an informal requirement. They neither start from a formal representation nor produce one. Generally, they provide a useful methodology for improving a person's thought processes [97, 98] but since they do not produce a formal representation they are not as useful for further computer-based acquisition or analysis. An analyst might use the RA to formalize a requirement so elicited (equivalently, an analyst might prepare to use the RA by engaging in such an elicitation process).

In his PhD thesis [99, 100], Leite describes *viewpoint resolution* as a methodology for gathering all of the necessary raw facts of a problem. Elicitation is done from different viewpoints and then the viewpoints are cross-checked for consistency and completeness. This serves to establish the problem's *universe of discourse.* The three viewpoints he uses are data centered, process centered, and actor centered. Basically, the methodology consists of thinking of the data objects needed, then thinking of the processes that manipulate them, then thinking of the actors that perform them, all while thinking about what other data each process manipulates, and what other processes an actor performs, and so on. This process is executed until closure. An advantage of this methodology is that it does not rely on any previously encoded domain knowledge, but it does rely on the intelligence of the analyst.

Leite has written a program to compare requirements acquired in this manner that are stated using a rule-based representation. Two viewpoints are each represented as a set of rules. The analysis finds pairs of similar rules. For each pair it finds, it identifies discrepancies between the two rules (missing patterns or attributes). Rules that have no matching partner are identified as missing in the other model.

Requirements may be written by one or more authors from any or all of the three viewpoints. Something of an N-version programming approach may be achieved by comparing the requirements created by different authors using one or more viewpoints.

Leite calls this a method for very early validation of a requirement.

Structured Design Methodologies

A number of structured design methodologies were introduced in the late 70's. Methods by DeMarco, Yourdon, Gane and Sarson, and Jackson are popular in the field. The IEEE Computer special issue on Requirements Engineering Environments [101] provides a good introduction to some of these methodologies.

One of the popular methodologies is SADT [102]. SADT is a visual discipline of boxes and arrows and viewpoints (data and process) that permits a hierarchical description of a system, its data, activities, and interfaces. SADT provides a mind discipline for the elicitation of requirements. This description applies pretty well to most design methodologies. Methodologies help the user to organize thoughts and can help produce effective communication artifacts. Unfortunately, they generally do not have a strong computer-based semantics since they operate at the most abstract level of thoughts and informality. Computer support or analysis is generally hard because these methodologies are so broad.

6.3 Classification

Related work on classification is divided into two subsections. The first, describes applications that use a classification algorithm to perform a recognition task. The Listener uses classification in this manner to recognize cliché instances. The second section briefly discusses how the classification algorithm used by the Listener relates to the standard algorithm described in the literature.

Applications

Consul [103, 104, 105] is a system that supports a cooperative interaction between users and computer tools, e.g., mail systems. Consul processes natural language requests for the execution of an action in the underlying computer system. The request is translated into a form understandable in terms of Consul's operational model of system activity. The domain model that facilitates this consists of a description of: users and what they do, systems, services, and the current dynamic system environment. The goal of the Consul system is to resolve informalities in the user's request by discovering an operational form for the request.

Consul achieves its goals by classifying input activity descriptions until they are an instance of a class with an operational definition. When classification terminates before obtaining an operational description, Consul searches for a best match operational description. It then attempts to apply action and object transformation rules that will eliminate the discrepancies preventing an exact match. Rule selection is triggered by classification [106] of components of the current description as an instance of a rule's left-hand-side pattern. The whole process is one of narrowing-down and homing-in, similar to the process used by the Listener in applying domain knowledge.

Consul also incorporates an acquisition component that uses classification to acquire new service definitions into the knowledge-base. Users interactively provide an initial classification of a newly defined service and are engaged in a dialog that requires them to align features of the service definition with pre-existing features in the knowledge-base.

LaSSIE [107] is an information system that documents the software of a large system, the AT&T System 75. It helps a programmer discover information about the structure and components of the program. This supports both program understanding and facilitates software reuse. LaSSIE uses a classification algorithm to support semantic retrieval of information about the target system.

A natural language front-end supports informal programmer queries. Queries may take a functional, architectural, feature or code view. The System 75 code has been described and indexed according to its properties and formally represented in a knowledge-base. The basic ontology used to describe the system includes the categories: object, action, doer, and state. Concepts at the leaves of the hierarchy point to system code files. Classification ensures that individual component descriptions are placed in their proper place in the knowledge-base. Queries are classified in the system description knowledge-base and the answers are the instances of the classified query. If desired, a query can be reformulated by generalization or specialization of features and reclassified.

The Classification Algorithm

Classification is a well known AI technique. Generally [4, 108, 109] it has been used as a process to precompile the subsumption relation in a monotonically increasing set of concepts. This precompilation process has also been used at runtime to classify data specific to a problem (as described in the systems above). This data is defined as a concept, but often one that is quite specific, e.g., "a message with recipient Smith and sender Jones." Classification is used both on concepts that denote a set (of cardinality greater than one) of individuals and on individual concepts that describe a set with a single element (called an instance in some cases).

In KL-ONE style representation systems, classification relies on the separation of terminological (intensional) and assertional (extensional) information. KL-ONE style classification operates based only on terminological information. This might be termed static classification since it reasons with stable terminological properties and not with changing assertional properties. Generally, this sort of classification is used when entering information in a large concept space in which it is difficult to be sure of what all the possible appropriate initial descriptions (parents) might be. For example, the concept of neurosurgeon might be indexed under: doctor, rich-person, patient-person, well-insured-person.

Static classification is inappropriate to the problems addressed by the Listener since the Listener must take into account contingent situational information. Most of this information cannot even be expressed in the purely terminological component of KL-ONE style systems and thus cannot participate in static classification or realization [110]. There are two additional considerations which make static classification inappropriate in the Listener system. First, there is no strict division of terminological and assertional information in the cliché definitions. Definitions may include non-terminological information expressed as, e.g., disjunctions, negations, or binary functions [111]. Second, it is assumed that all instances are initially indexed under an appropriate set of concepts, or that the Listener will deduce relevant orthogonal initial concepts via some other processing.

The Listener uses the changing assertional state to drive its classification process. This might be termed dynamic classification. Dynamic classification requires an initial concept indexing to provide a candidate set of classifications. If the initial set of candidates is incomplete, the Listener will miss a recognition opportunity.

Whereas static classification is a central process in KL-ONE based systems, dynamic classification is central to the Listener system. The Listener is not required to perform general discovery, rather its goal is to form the best description it can within the constraints of what the speaker has said. This permits the dynamic classification routine to explore a limited set of candidates more completely.

Chapter 7

Future Work and Conclusions

The problem of acquisition of formal knowledge from informal input sources is replete with problems to be solved. This discussion of future work highlights a few of the logical next steps in developing a Listener system. The three areas addressed are: dealing with the limitations of the underlying reasoning system, presenting contradictions in a manner that makes the underlying problem clear, and developing a more realistic requirements acquisition system.

Limited Reasoning and Limited Resources

CAKE is a limited logical reasoning system. For example, it does not perform complete propositional reasoning. Even with these limitations, which permit the reasoning system to run relatively quickly, the Listener requires a significant amount of execution time. The RA requires a few minutes to process most commands whereas a practical system would have to complete this processing much quicker. These limitations effect the Listener in at least two areas.

Recall that the Listener may suggest possible repairs to fix a conflict, e.g., alternate types for an instance. In principle, the Listener's reasoning abilities are capable of evaluating these suggestions, ordering them, and eliminating contradictory suggestions or suggestions that do not actually solve the problem. To date, support for this kind of processing, i.e., hypothetical reasoning, has not been pursued because it would undoubtedly require too much computation time to be practical in an interactive system. However, this kind of computation is an ideal candidate for background processing. It is plausible that the Listener could carry out such investigations overnight and greet the analyst in the morning with an improved set of suggestions.

Other experiments that might be done with the reasoning system include providing some sort of power dial which would allow trading time spent in reasoning for completeness of the reasoning procedure [112]. Also, it may be feasible to run a more complete reasoning procedure overnight. It would be interesting to observe what useful additional deductions this effort yields.

Another limitation of the Listener system is its use of first-order representations and algorithms. The RA simulates some second-order deductions regarding statements being "possible" and "not tautologous." For example, in the air traffic control example there are a number of "error-when" condition statements. The error predicate must not be statically true or false (i.e., the error always occurs or never occurs) rather it must simply be possible for it to occur (in this case it must remain unknown). Need statements should not be tautologous, otherwise they are simply statements of truth that do not need to be implemented in the eventual system. But asserting a Need asserts the truth of the desired condition by default (so that need interactions can be analyzed). Therefore, detecting a tautology requires retracting the desired condition, checking for a residual truth value, and reasserting the desired condition. Adopting some second-order reasoning algorithms would open up consideration of more of these sorts of conditions.

Understanding Contradictions

The explanations produced by the reasoning system for a contradiction are quite large and cryptic. The size is due partially to the explicitness of the underlying logic. For example, CAKE will form the intermediate node `(And A B C)` if it is part of the support of another node and it will report as the explanation of `(And A B C)` the fact that `A` is true and `B` is true and `C` is true. For a person observing such an explanation, this is redundant (no explanation is really needed) and takes up twice as much space in a graphical explanation tree. Or, for example, if `A` is true and `(Implies A B)` is true, then the explanation of `B` includes both nodes (versus, for example, just the node `A` with an "implies" label on the arc that designates `A` is a support for `B`.) Again, this takes up twice as much space, which can be significant when `A` and `B` are large structured assertions.

Lefelhocz [28] has developed an explanation grapher that prunes the explanation tree by eliminating redundant logical explanations. The grapher can also be told to ignore internal bookkeeping premises which further reduces the size of the explanation. This grapher goes part of the way to reducing explanations to a comprehensible size.

Still, this leaves an unordered set of premises as candidates for resolving a contradiction. An area where some interesting work might be done is trying to automate a tool that gets at the heart of the contradiction. Certain premises are more likely candidates for change than others. Factors involved in this evaluation include recency of the assertion and the number of other assertions supported by it, i.e., how much of a commitment there is to that particular assertion.

More Comprehensive Requirements Methodology

The RA is generally passive and non-invasive. It does not embody any particular acquisition methodology. The analyst controls the interaction with minimum guidance from the RA, e.g., the agenda of pending issues. Two modes might be added to the RA, and to the Listener in general, to facilitate acquisition. An elicitation mode might be added that would prompt the analyst to answer a number of exhaustive questions based on the structure of the problem domain. This is a common knowledge acquisition technique. Also, an analysis mode might be added. In this mode, traceability of requirements to specification might be acquired, e.g., for each Need stated there must be a part of the System that "satisfies" that Need (and each component of the System should be present to satisfy some Need).

Including a more aggressive classification strategy would also be consistent with the desire for a more interactive methodology. It may not be realistic to expect 100 percent satisfaction of certain cliché definitions or even the stateability of such definitions. The analyst could be led through an exploration of the space of possible descriptions favoring clichés that are clearly likely choices, or the only remaining choice.

Currently, the analyst makes decisions regarding what to say and how to resolve conflicts and then simply makes the appropriate statements. No record is kept of the deliberation that leads to these decisions. No analysis is recorded of the various issues that are raised, arguments put forward, and positions that are adopted [93, 94]. Neither are the facts that support these issues, positions, and arguments recorded. A place exists for the representation of such a deliberation process and the integration of that process with the underlying facts represented in the DKB.

Finally, in the area of requirements analysis, the RA overlooks the possibility that the description could be too specific, a potentially bad feature of a requirement. The RA might also consider generalizing statements to see what information is lost. This, of course, has to be balanced against the general strategy of applying the most specific applicable clichés to a situation.

7.1 Conclusions

The Listener system has been demonstrated in the domain of legal document acquisition and software requirements acquisition. The same representation technology and reasoning tools proved useful in both applications. This breadth provides some preliminary evidence that these techniques are generally useful as knowledge acquisition tools.

Research on the RA addresses an important gap in much of the work on tools to support requirements analysis. Whereas the requirements process almost inevitably begins with only a vague and informal statement of what is desired, most requirements analysis tools need some sort of relatively formal requirement statement as their input. The RA attacks head-on the problem of fleshing out and cleaning up a vague and informal requirement statement.

The current version of the system demonstrates a number of capabilities that can be used to bridge the gap between informal and formal specifications. The RA has a knowledge-base of requirements clichés that allows it to infer large amounts of information from the analyst's statements. The analyst is able to talk in terms of high-level terms leaving the task of disambiguating these terms to the RA. The RA continually checks the requirement for consistency (searching for contradictions) and completeness (based on expectations set up by various clichés). The RA includes specific support for evolutionary modification of the requirement, using the facilities of the underlying reasoning system to ensure that each change is carried out in a consistent way throughout the requirement.

The partnership between the RA and an analyst embodies an effective division of labor where the analyst is responsible for communicating with the end user and entering the core information in the requirement and the RA takes care of the details making sure that the requirement is always in a consistent state even after numerous modifications.

Appendix A

Requirements Document Reader's Guide

Requirements Document Reader's Guide Revision 2.1

Howard B. Reubenstein

June 12, 1990

The attached requirements document was automatically produced by the Requirements Apprentice (RA). This guide explains the syntactic and organizational conventions used in its production. An automatically produced document is precise in a way that is difficult and tedious to achieve when writing a document by hand. Term usage, layout, and level of detail all follow the algorithm defined for document production. Thus, the reader can make reliable inferences based on detailed features of the actual presentation. This increases the information content of the document. The reader can rely on the RA's precise following of these conventions as an aid to understanding the document.

The document consists of: a title page, table of contents, introduction, environment, needs and system chapters, and the requirement's state chapter. The environment, needs, and system chapters each include a dictionary of terms used in that section. This decomposition defines the basic, fixed, top level structure of a requirements document. Additional structure may be added within this framework during creation of the document.

The introduction is generated by creating a top level description of the environment and system objects based on the clichés they are instances of. Each cliché includes various levels of canned-text documentation, among them are the *abstract* and *overview* levels. The abstract level is used in generating material for the introduction, whereas the overview level is used in succeeding, more detailed, descriptions in the following chapters.

The body of requirement is described in sections 2, 3 and 4. Section 2 describes the *environment* that the system to be built has to operate in. It describes the "immutable" reality that exists in the world. Changes to this section reflect a drastic change in the view of the world.

The environment section and the system section are created by the same mechanism. Environments and systems are basically the same kind of things. It is mainly perspective that differentiates them. This can be seen by considering the following. One requirement's system may be another's environment. For example, banking environments have required ATM systems to be developed. The ATM system is the environment for the ATM software system.

The environment section starts with a detailed description of environment objects. This description begins with the overview level of canned text associated with the most specific types of an environment object. Then the roles of the object are enumerated and described. If a role has a value, then this value is presented. If the role has no value, but it has a type, then the role is documented as an instance of the appropriate type. If a role has a value or a type, then the static role documentation, describing the purpose of the role, is printed. If a role has neither a value or type, it appears in the list of unfilled roles.

Certain roles on clichés are designated as worthy of description in separate sections. For example, on the high level clichés *environment* and *system* there are roles that hold information about actions, agents, and behaviors that are printed in separate sections. In information systems, reports are printed in a separate section. Document sections for such roles are added on the fly and fit into the basic fixed top level document structure.

Section 3 describes the *needs* that the system has to meet. It defines a viewpoint that captures the perceived problem in the environment. The needs section includes a description of all the needs (problems to be solved, policies to be enforced, and product functions to be supported) that the system has to address. The needs define the problem in the environment that the system to be built has to solve. The method of description is basically the same as in the environment and systems section. Needs objects tend to be less structured, however, therefore the descriptions are generally shorter.

Section 4 describes the *system* that actually will be built to address the problem defined in the previous two sections. The system description in this document is an initial very high level specification.

Each of sections 2, 3, and 4 has a dictionary subsection. These dictionaries describe technical terms used in the section. Certain technical terms (those defined in a reference library of clichés), e.g., system or need, are not included in the dictionary under the assumption that they will be documented in a user's manual.

Section 5, the requirement's state chapter, is produced in recognition of the fact that a requirement is rarely complete. It captures the state of the requirements analysis describing the loose ends in the current state of the requirements acquisition. The loose ends include:

- any significant true facts that did not make their way into the document. These are discovered by searching through the knowledge base for assertions of a certain syntactic form (excluded are type assertions and equalities which will have been included in the document) that would not have been included in the requirements document due to the nature of the object-type-role centered traversal of the RKB.

- a list of retracted defaults and other retracted decisions
- the pending agenda items
- a list of roles that were queried about in creating the document that did not have values
- any outstanding contradictions

The document uses certain syntactic conventions:

- A compound word Foo.Bar.Baz means the Baz property of the Bar property of the object Foo.
- The use of "(unknown)" after a dotted compound word indicates that the value of that property is unknown.
- Some section titles are generated automatically, e.g., Policy-100, when the full name would be too long.
- The dictionary section has an entry for each object discussed in the preceding section text, except for those that are defined in the reference library. Terms in the reference library are common to any requirements analysis performed by the RA. Their definition can be found in the RA reference manual.

The document is generated by a structured walk through of the RKB. Object descriptions are generated by a combination of enumerating the types of an object, describing its roles, and documenting the object according to canned-text associated with the various clichés of which it is an instance.

Appendix B

An Automatically Generated Requirements Document

Library-System Requirement

[Working Draft 1]

HBR

5/02/90 16:50:49

Contents

1 Introduction

This is the library-system requirement. The problem is set in an environment described as the university-library (UL). The university-library-database (ULDB) provides the solution.

UL is a grouped-lending-repository.

The UL stores copy-of-books organized as groups of books. Copy-of-books that are in the repository may be borrowed for a finite period of time by the patrons. The repository's collection may shrink or grow in size as it is maintained by the staff.

ULDB is an advisory-system and tracking-information-system.

The ULDB regulates the ULDB.system-monitored (unknown) using notifications and authorizations to ULDB.advice-recipients (unknown). The ULDB monitors and attempts to enforce a set of restrictions placed on a target physical system. The enforcement power of the ULDB is, however, purely advisory.

The ULDB assists the patrons and staff of the UL by tracking copy-of-books. It maintains an internal image of the changing physical state of the UL. Reports may be generated from this internal image to summarize the state of the UL.

There are 12 statements that define the system needs. They are divided into 3 categories: polices (6 statements), product functions (5), and needs (1). The following more specialized need types are used in the definition of some of the 12 needs statements: action-authorization-policy, privacy-policy, and resource-control-policy.

See the accompanying reader's guide for assistance in interpreting this document.

2 Environment

The university-library (UL) is a grouped-lending-repository.

The UL is a repository of objects of type copy-of-book. These objects are grouped into classes of book in which individual objects are considered interchangeable. The staff of the repository regulates the collection by deciding when objects in the collection need to be removed or added. The patrons of the repository borrow and return items in the collection.

The roles of the UL are:

- Collection-type: copy-of-book.

  is the type of physical entities in the collection.

- Group-type: book.

  is the type of abstract entities into which collection members are grouped.

- Item-states: lent-out, missing, and stored.

  describes the possible states that component items may be in.

  Copy-of-book that are members of the UL may be in one of three states: stored, lent-out, and missing. This trinary model is based on the assumption that items to be lent may be in, out, or unaccounted for. This model does not make distinctions between the ways in which an item may be unaccounted for.

- Access-limits: unknown.

  describes limits on the execution of certain actions by certain privileged or non-privileged groups.

- Borrowing-period: finite.

The unfilled slots are state-of.

Book is a type. Its parent types are bag. Book has local slots author, isbn, title, and topic. Book inherits 2 other slots from its parents. Book is a group of copy-of-book.

Copy-of-book is a type. Its parent types are physical-object. Copy-of-book has local slots author, isbn, and title. Copy-of-book inherits 1 other slots from its parents.

Library is a type. Its parent types are repository. Library has no local slots. Library inherits 11 other slots from its parents.

2.1 UL Actions

This section describes the possible state transition operators.

2.1.1 Add-group-grouped-repository

Add-group-grouped-repository is a grouped-repository-state-modification-action.

Add-group-grouped-repository adds a new group of items to the collection. This corresponds to an add-repository action where the class of item added did not previously exist in the repository.

- input: (?The-agent ?Item).

- pre: (And (Book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent))).
- normal-pre: (Not (Mem-repository ?Item UL)).
- normal-post: (Mem-repository ?Item UL).
- class-type: book.
- item-type: copy-of-book.

The unfilled slots are post and normal-effects.

2.1.2 Add-repository

Add-repository is a repository-state-modification-action.

Add-repository adds an item to the collection of the UL. Each addition is made by a particular agent. Normally, the result of the addition will be the membership of the item in the repository.

- input: (?The-agent ?Item).
- pre: (And (Copy-of-book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent))).
- normal-pre: (Not (Mem-repository ?Item UL)).
- normal-post: (Mem-repository ?Item UL).
- item-type: copy-of-book.

The unfilled slots are post and normal-effects.

2.1.3 Borrow-lending-repository

Borrow-lending-repository is a repository-state-modification-action.

Borrow-lending-repository removes an item from the collection for a finite period of time. The item becomes inaccessible and responsibility for the item is assigned to the borrower. In order for the item to be borrowed it must initially be stored and accessible in the repository.

- input: (?The-borrower ?Item).
- pre: (And (Copy-of-book ?Item) (Patrons ?The-borrower)).
- normal-pre: stored-tri-state-model.
- normal-effects: located-out.
- normal-post: lent-out-tri-state-model.
- item-type: copy-of-book.

The unfilled slots are post.

2.1.4 Remove-group-grouped-repository

Remove-group-grouped-repository is a grouped-repository-state-modification-action.

Remove-group-grouped-repository permanently removes a group of items from the collection. This corresponds to a remove-repository action where the item removed is the last of its class in the repository.

- input: (?The-agent ?Item).
- pre: (And (Book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent))).
- normal-pre: (Mem-repository ?Item UL).
- normal-post: (Not (Mem-repository ?Item UL)).
- class-type: book.
- item-type: copy-of-book.

The unfilled slots are post and normal-effects.

2.1.5 Remove-repository

Remove-repository is a repository-state-modification-action.

Remove-repository permanently removes an item from the collection of the UL. Each removal is made by a particular agent. The item must be in the collection before it can be removed. An equivalent item may be added back at a later time.

- input: (?The-agent ?Item).
- pre: (And (Copy-of-book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent))).
- normal-pre: (Mem-repository ?Item UL).
- normal-post: (Not (Mem-repository ?Item UL)).
- item-type: copy-of-book.

The unfilled slots are post and normal-effects.

2.1.6 Return-lending-repository

Return-lending-repository is a repository-state-modification-action.

Return-lending-repository restores an item that was previously borrowed by an agent. An unresolved issue is whether the returning agent must be the same as the borrowing agent. Normally the item returned is expected to have been borrowed from the lending-repository, however, it is possible that a lost or stolen item will be returned. The result of the operation is to bring the item back into the lending-repository to eventually be ready for borrowing.

- input: (?The-borrower ?Item).
- pre: (Copy-of-book ?Item).
- normal-pre: lent-out-tri-state-model.

- normal-effects: located in.
- normal-post: stored-tri-state-model.
- item-type: copy-of-book.

The unfilled slots are post.

2.2 UL Agents

Agents generate activity by performing actions. The agents are: patrons and staff.

UL.patrons are the primary interacting agents.

UL.patrons are patrons of the repository. Patrons engage the services of the repository. Patrons executes the borrow-item function of UL.

UL.staff are staff of the Repository. The staff of a repository is responsible for managing the collection and ensuring its integrity and availability. Staff executes the maintain-repository function of UL.

2.3 Dictionary

- **Access-policy** is a type. Its parent types are execution-access-policy. Access-policy has local slots privileged-group and restricted-action. Access-policy inherits 3 other slots from its parents.
- **Add-group-grouped-repository-action** is a type. Its parent types are action and grouped-repository-state-modification-action. Add-group-grouped-repository-action has no local slots. Add-group-grouped-repository-action inherits 13 other slots from its parents.
- **Add-repository-action** is a type. Its parent types are action and repository-state-modification-action. Add-repository-action has no local slots. Add-repository-action inherits 12 other slots from its parents.
- **Borrow-lending-repository-action** is a type. Its parent types are action and lending-repository-state-modification-action. Borrow-lending-repository-action has no local slots. Borrow-lending-repository-action inherits 13 other slots from its parents.
- **Grouped-lending-repository** is a type. Its parent types are grouped-repository and lending-repository. Grouped-lending-repository has no local slots. Grouped-lending-repository inherits 15 other slots from its parents.
- **Grouped-repository-state-modification-action** is a type. Its parent types are action and repository-state-modification-action. Grouped-repository-state-modification-action has local slots class-type and home-repository. Grouped-repository-state-modification-action inherits 11 other slots from its parents.
- **Lending-repository-state-modification-action** is a type. Its parent types are action and repository-state-modification-action. Lending-repository-state-modification-action has local slots home-repository and the-borrower. Lending-repository-state-modification-action inherits 11 other slots from its parents.
- **Lent-out** is a relation of arity 2. Lent-out is a relation between an object and a storage location. It is only defined on objects that are members of the storage location.

- **Lent-out-tri-state-model** is a relation of arity 2 of type constraint.
- **Missing** is a relation of arity 2.
- **Patrons** is an agent.
- **Patrons-agent-repository** is a type. Its parent types are people and patrons-agent.
- **Remove-group-grouped-repository-action** is a type. Its parent types are action and grouped-repository-state-modification-action. Remove-group-grouped-repository-action has no local slots. Remove-group-grouped-repository-action inherits 13 other slots from its parents.
- **Remove-repository-action** is a type. Its parent types are action and repository-state-modification-action. Remove-repository-action has no local slots. Remove-repository-action inherits 12 other slots from its parents.
- **Repository-state-modification-action** is a type. Its parent types are action. Repository-state-modification-action has local slots home-repository, item, item-type, and the-agent. Repository-state-modification-action inherits 8 other slots from its parents.
- **Return-lending-repository-action** is a type. Its parent types are action and lending-repository-state-modification-action. Return-lending-repository-action has no local slots. Return-lending-repository-action inherits 13 other slots from its parents.
- **Staff** is an agent.
- **Staff-agent-repository** is a type. Its parent types are people and staff-agent.
- **Stored** is a relation of arity 2. Stored is a relation between an object and a storage location. It is only defined on objects that are members of the storage location.
- **Stored-tri-state-model** is a relation of arity 2 of type constraint.
- **Tri-state-model** is a type. Its parent types are lending-state-model. Tri-state-model has no local slots. Tri-state-model inherits 2 other slots from its parents.
- **UL** (see section 2) is a grouped-lending-repository.

  The UL is a repository of objects of type copy-of-book. These objects are grouped into classes of book in which individual objects are considered interchangeable. The staff of the repository regulates the collection by deciding when objects in the collection need to be removed or added. The patrons of the repository borrow and return items in the collection.

3 Needs

3.0.1 (Tracks university-library-database university-library)

(tracks ULDB UL) is a need.
The unfilled slots are handled-by.

3.1 Policies

3.1.1 R1-add

R1-add is an action-authorization-policy.
R1-add is a policy which describes a criterion that the ULDB should use in deciding if an action should be permitted. ∀ patrons-agent-repository it is true that iff patrons-agent-repository executes add-item then the patrons-agent-repository must be in staff-agent-repository. This criterion describes appropriate action sequences without indicating the motivation for the restrictions.

- authorized-group: staff-agent-repository.
- monitor: ULDB.
- restricted-action: add-item.
- restricted-group: patrons-agent-repository.

The unfilled slots are handled-by and enforcer.

3.1.2 R1-remove

R1-remove is an action-authorization-policy.
R1-remove is a policy which describes a criterion that the ULDB should use in deciding if an action should be permitted. ∀ patrons-agent-repository it is true that iff patrons-agent-repository executes remove-item then the patrons-agent-repository must be in staff-agent-repository. This criterion describes appropriate action sequences without indicating the motivation for the restrictions.

- authorized-group: staff-agent-repository.
- monitor: ULDB.
- restricted-action: remove-item.
- restricted-group: patrons-agent-repository.

The unfilled slots are handled-by and enforcer.

3.1.3 R2

R2 is a privacy-policy.

R2 is policy which describes system privacy considerations. ∀ patrons-agent-repository it is true that iff patrons-agent-repository performs last-borrower-of-book then the patrons-agent-repository must be in staff-agent-repository.

- authorized-group: staff-agent-repository.
- restricted-action: last-borrower-of-book.
- restricted-group: patrons-agent-repository.

The unfilled slots are handled-by and enforcer.

3.1.4 R3

R3 is a privacy-policy.

R3 is policy which describes system privacy considerations. ∀ patrons-agent-repository it is true that iff patrons-agent-repository performs books-by-borrower then the patrons-agent-repository must be in (!the-set-of-all (?patron) s.t. (or (staff-agent ?patron) (= ?patron (user-target (perform ?patron books-by-borrower))))).

- authorized-group: (!the-set-of-all (?patron) s.t. (or (staff-agent ?patron) (= ?patron (user-target (perform ?patron books-by-borrower))))).
- restricted-action: books-by-borrower.
- restricted-group: patrons-agent-repository.

The unfilled slots are handled-by and enforcer.

3.1.5 Policy-1

Policy-1 is a policy.

It is a policy which describes resource availability considerations. This description applies to all instances of type resource-control-policy.

- value: (!text-need "A Borrower may borrow no more than <Book-Limit> Books at a time.").

The unfilled slots are handled-by and enforcer.

3.1.6 (!Text-need "A Borrower may not borrow two copies of the same book.")

(!Text-need "A Borrower may not borrow two copies of the same book.") is a policy.

(!Text-need "A Borrower may not borrow two copies of the same book.") is a resource-control-policy.

The unfilled slots are handled-by and enforcer.

3.2 Product Functions

3.2.1 Ensure-accuracy

Ensure-accuracy is a need
The unfilled slots are handled-by.

3.2.2 Generate-information-reports

Generate-information-reports is a need.
The unfilled slots are handled-by.

3.2.3 Monitor-and-authorize

Monitor-and-authorize is a need.
The unfilled slots are handled-by.

3.2.4 Track-observed-data

Track-observed-data is a need.

- handled-by: (track-state-changes ULDB).

3.2.5 Track-target-state

Track-target-state is a need.
The unfilled slots are handled-by.

3.3 Dictionary

- **Action-authorization-policy** is a type. Its parent types are policy and restriction. Action-authorization-policy has local slots authorized-group, restricted-action, and restricted-group. Action-authorization-policy inherits 4 other slots from its parents.
- **Add-item** is a behavior. Add-item is built upon the Add action. Generally the purpose of adding an item to the repository is to improve its stock and thus improve its ability to serve its clients.
- **Privacy-policy** is a type. Its parent types are policy. Privacy-policy has local slots authorized-group, restricted-action, and restricted-group. Privacy-policy inherits 3 other slots from its parents. Privacy-policy describes limits on the availability of information.
- **R1-add** (see section 3.1.1) is an action-authorization-policy.

  R1-add is a policy which describes a criterion that the ULDB should use in deciding if an action should be permitted. $\forall$ patrons-agent-repository it is true that iff patrons-agent repository executes add-item then the patrons-agent-repository must be in staff-agent-repository. This criterion describes appropriate action sequences without indicating the motivation for the restrictions.

- **R1-remove** (see section 3.1.2) is an action-authorization-policy.

  R1-remove is a policy which describes a criterion that the ULDB should use in deciding if an action should be permitted. ∀ patrons-agent-repository it is true that iff patrons-agent-repository executes remove-item then the patrons-agent-repository must be in staff-agent-repository. This criterion describes appropriate action sequences without indicating the motivation for the restrictions.

- **R2** (see section 3.1.3) is a privacy-policy.

  R2 is policy which describes system privacy considerations. ∀ patrons-agent-repository it is true that iff patrons-agent-repository performs last-borrower-of-book then the patrons-agent-repository must be in staff-agent-repository.

- **R3** (see section 3.1.4) is a privacy-policy.

  R3 is policy which describes system privacy considerations. ∀ patrons-agent-repository it is true that iff patrons-agent-repository performs books-by-borrower then the patrons-agent-repository must be in (!the-set-of-all (?patron) s.t. (or (staff-agent ?patron) (= ?patron (user-target (perform ?patron books-by-borrower))))).

- **Remove-item** is a behavior. Remove-item is built upon the Remove action. There are two purposes for removing items. First is to support the consumption function of the repository. Second is to remove inventory that is damaged or inappropriate for use.

- **Resource-control-policy** (see section 3.1.5) is a type. Its parent types are policy. Resource-control-policy has no local slots. Resource-control-policy inherits 3 other slots from its parents. Resource-control-policy describes limits on the allocation of resources.

- **Restriction** is a type. Its parent types are policy. Restriction has local slots monitor. Restriction inherits 3 other slots from its parents.

4 System

The university-library-database (ULDB) is an advisory-system and tracking-information-system.

The ULDB monitors the ULDB.system-monitored (unknown) restriction policies and issues the appropriate authorizations. The ULDB will observe requests from users and acknowledge these requests by either confirming the request and issuing an Authorization or by denying the request and issuing advice in the form of a Notification. The user is expected to adhere to the advice given by the system. The advisory system operates under an assumption that users are benevolent and simply do not always realize that their actions may be in violation of policy.

The main purpose of ULDB is to track the state of physical entities. patrons and staff act in the environment causing changes to the state of the university-library (UL) which are reflected in ULDB. The ULDB attempts to maintain an image of the state of UL by processing requests issued by Users. The ULDB maintains a catalog of records which mirrors the state of the tracked target. These records may be created and destroyed and may be used to generate informational reports.

The roles of the ULDB are:

- Item-states: available and checked-out.

  describes the possible states that component items may be in.

  ULDB.collection-type (unknown) that are members of the ULDB may be in one of two states available and checked-out. This binary model is based on the simple assumption that items to be lent may either be In or Out. No provision is made for any intermediate states.

- Target: UL.

  The UL is a repository of objects of type copy-of-book. These objects are grouped into classes of book in which individual objects are considered interchangeable. The staff of the repository regulates the collection by deciding when objects in the collection need to be removed or added. The patrons of the repository borrow and return items in the collection.

- Target-state: lent-out, missing, and stored.

- The-data-observed: unknown.

  ULDB.the-data-observed is state-changes.

- The-manner-of-observation: unknown.

  ULDB.the-manner-of-observation is indirect-observation.

The unfilled slots are tracked-state, system-restrictions, system-monitored, options, advice-recipients, and additional-information.

ULDB executes the track-state function.
ULDB executes the regulate function.
ULDB executes the authorize function.
The next section(s) describe the following slot(s): functional-requirements and reports, that are important enough to merit individual sections.

4.1 Functional-Requirements of ULDB

The functional requirements of a System are a set of operations that the system must support in order to satsify the defined needs of the system. For purposes of validation functional requirements should be traceable to specific needs and every need should be accounted for by one or more functional requirements

4.1.1 Acquisition

Acquisition is an action-tracking-operation.
Acquisition records the add-repository of copy-of-book objects from UL by acquisition.the-agent (unknown).

- input: (?Operator ?The-agent ?Item).
- pre: (And (Copy-of-book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent))).
- normal-pre: (Not (Mem-repository ?Item UL)).
- normal-post: (Mem-repository ?Item UL).
- records: add-repository.
- target: UL.
- object-type: copy-of-book.
- objects: (!the-set-of-all (?o) s.t. (= ((!cross isbn copy-num) ?o) input)).

The unfilled slots are post and normal-effects.

4.1.2 Check-in

Check-in is an action-tracking-operation.
Check-in records the return-lending-repository of copy-of-book objects from UL by check-in.the-agent (unknown).

- input: (?Operator ?The-borrower ?Item).
- pre: (Copy-of-book ?Item).
- normal-pre: lent-out-tri-state-model.
- normal-effects: located-in.
- normal-post: stored-tri-state-model.
- records: return-lending-repository.
- target: UL.
- object-type: copy-of-book.
- objects: (!the-set-of-all (?o) s.t. (= ((!cross isbn copy-num) ?o) input)).

The unfilled slots are post.

4.1.3 Check-out

Check-out is an action-tracking-operation.
Check-out records the borrow-lending-repository of copy-of-book objects from UL by check-out.the-agent (unknown).

- input: (?Operator ?The-borrower ?Item).
- pre: (And (Copy-of-book ?Item) (Patrons ?The-borrower)).
- normal-pre: stored-tri-state-model.
- normal-effects: located-out.
- normal-post: lent-out-tri-state-model.
- records: borrow-lending-repository.
- target: UL.
- object-type: copy-of-book.
- objects: (!the-set-of-all (?o) s.t. (= ((!cross isbn copy-num) ?o) input)).

The unfilled slots are post.

4.1.4 Unshelf

Unshelf is an action-tracking-operation.
Unshelf records the remove-repository of copy-of-book objects from UL by unshelf.the-agent (unknown).

- input: (?Operator ?The-agent ?Item).
- pre: (And (Copy-of-book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent))).
- normal-pre: (Mem-repository ?Item UL).
- normal-post: (Not (Mem-repository ?Item UL)).
- records: remove-repository.
- target: UL.
- object-type: copy-of-book.
- objects: (!the-set-of-all (?o) s.t. (= ((!cross isbn copy-num) ?o) input)).

The unfilled slots are post and normal-effects.

4.1.5 Unshelf-all

Unshelf-all is an action-tracking-operation.
Unshelf-all records the remove-repository of copy-of-book objects from UL by unshelf-all.the-agent (unknown).

- input: (?Operator ?The-agent ?Item).
- pre: (And (Copy-of-book ?Item) (Or (Staff ?The-agent) (Patrons ?The-agent))).
- normal-pre: (Mem-repository ?Item UL).
- normal-post: (Not (Mem-repository ?Item UL)).
- records: remove-repository.
- target: UL.
- object-type: copy-of-book.
- objects: (!the-set-of-all (?o) s.t. (= (isbn ?o) input)).

The unfilled slots are post and normal-effects.

4.2 Reports of ULDB

The reports of an Information-System are a special class of transactions. Their purpose is to generate information about the state of the system without altering that state. Reports are described by an *input data signature*, an output data signature, and a query which defines the functionality of the report.

4.2.1 Books-by-author

Books-by-author is an information-system-report and tracking-report.
Books-by-author is a report which provides information about the state of the information system. This description applies to all instances of type information-system-report.
Books-by-author is a report which summarizes what the current state of the target objects is believed to be. This description applies to all instances of type tracking-report.

- accessed-information: Author-of-Book, Title-of-Book.
- data-in: (!decl ((?a author))).
- data-out: (!decl ((?titles (!set-of title)))).
- query:
  (= ?titles (!the-set-of-all (?t title) s.t. (∃ (?b book) s.t. (and (mem ?b UL) (= ?t ?b.title) (= ?a ?b.author))))).
- target: UL.

The unfilled slots are purpose and implementation.

4.2.2 Books-by-borrower

Books-by-borrower is an information-system-report and tracking-report.

- accessed-information: none.
- data-in: (!decl ((?b borrowers))).
- data-out: (!decl ((?books (!set-of book)))).
- query:
  (= ?books (!the-set-of-all (?a-book book) s.t. (borrower ?a-book UL ?b))).
- target: UL.

The unfilled slots are purpose and implementation.

4.2.3 Books-by-topic

Books-by-topic is an information-system-report and tracking-report.

- accessed-information: Topic-of-Book, Title-of-Book.
- data-in: (!decl ((?topic topic))).
- data-out: (!decl ((?titles (!set-of title)))).
- query:
  (= ?titles (!the-set-of-all (?t title) s.t. (∃ (?b book) s.t. (and (mem ?b UL) (= ?t ?b.title) (= ?topic ?b.topic))))).
- target: UL.

The unfilled slots are purpose and implementation.

4.2.4 Last-borrower-of-book

Last-borrower-of-book is an information-system-report and tracking-report.

- accessed-information: none.
- data-in: (!decl ((?a-book copy-of-book))).
- data-out: (!decl ((?b borrowers))).
- query:
  (!if (∃ (?a-borrower) s.t. (borrower ?a-book UL ?a-borrower)) (= ?b ?a-borrower) (last-borrower ?a-book UL ?b)).
- target: UL.

The unfilled slots are purpose and implementation.

4.3 ULDB Actions

This section describes the possible state transition operators.

4.3.1 Approve-advisory-system

Approve-advisory-system is an operation*.
Approve-advisory-system is the action taken to authorize a request.
The unfilled slots are post, normal-post, normal-effects, normal-pre, pre, and input.

4.3.2 Notify-advisory-system

Notify-advisory-system is an operation*.
Notify-advisory-system is the action taken to deny a request.
The unfilled slots are post, normal-post, normal-effects, normal-pre, pre, and input.

4.3.3 Track-target-tracking-system

Track-target-tracking-system is an operation*.
Track-target-tracking-system is the action taken to acquire an image of the tracked system.
The unfilled slots are post, normal-post, normal-effects, normal-pre, pre, and input.

4.4 ULDB Agents

Agents generate activity by performing actions. The agents are: users.
ULDB.users are the primary interacting agents.
ULDB.users are users of the Information-System. The users access the data stored in the system, typically invoking report functions or issuing requests for state update. Users executes the issue-request function of ULDB.
Users executes the obtain-information function of ULDB.
Users executes the produce-state-report function of ULDB.

4.5 Dictionary

- **Acquisition** (see section 4.1.1) is an action-tracking-operation.

  Acquisition records the add-repository of copy-of-book objects from UL by acquisition.the-agent (unknown).

- **Action-tracking-operation** is a type. Its parent types are transaction.
  Action-tracking-operation has local slots object-type, objects, records, records-function, target, and the-agent. Action-tracking-operation inherits 8 other slots from its parents.

- **Add-repository** is an action of type action-name and operation*.

  Add-repository is a pf-type.

  The unfilled slots are post, normal-post, normal-effects, normal-pre, pre, and input.

- **Advisory-system** is a type. Its parent types are system. Advisory-system has local slots advice-recipients, system-monitored, and system-restrictions. Advisory-system inherits 4 other slots from its parents.

- **Approve-advisory-system-action** is a type. Its parent types are action. Approve-advisory-system-action has no local slots. Approve-advisory-system-action inherits 8 other slots from its parents.
- **Available** is a relation of arity 2.
- **Binary-state-model** is a type. Its parent types are lending-state-model. Binary-state-model has no local slots. Binary-state-model inherits 2 other slots from its parents.
- **Books-by-author** (see section 4.2.1) is an information-system-report and tracking-report.

  See section 4.2.1 for a more detailed description of information-system-report.

  See section 4.2.1 for a more detailed description of tracking-report.
- **Books-by-borrower** (see section 4.2.2) is an information-system-report and tracking-report.

  See section 4.2.1 for a more detailed description of information-system-report.

  See section 4.2.1 for a more detailed description of tracking-report.
- **Books-by-topic** (see section 4.2.3) is an information-system-report and tracking-report.

  See section 4.2.1 for a more detailed description of information-system-report.

  See section 4.2.1 for a more detailed description of tracking-report.
- **Borrow-lending-repository** is an action of type action-name and operation*.

  Borrow-lending-repository is a pf-type.

  The unfilled slots are post, normal-post, normal-*effects*, normal-pre, pre, and input.
- **Check-in** (see section 4.1.2) is an action-tracking-operation.

  Check-in records the return-lending-repository of copy-of-book objects from UL by check-in.the-agent (unknown).
- **Check-out** (see section 4.1.3) is an action-tracking-operation.

  Check-out records the borrow-lending-repository of copy-of-book objects from UL by check-out.the-agent (unknown).
- **Checked-out** is a relation of arity 2.
- **Indirect-observation** is a type. Its parent types are manner-of-observation. Indirect-observation has local slots intermediary. Indirect-observation inherits 1 other slots from its parents.
- **Information-system-report** (see section 4.2.1) is a type. Its parent types are report. Information-system-report has local slots implementation and purpose. Information-system-report inherits 4 other slots from its parents.
- **Last-borrower-of-book** (see section 4.2.4) is an information-system-report and tracking-report.

  See section 4.2.1 for a more detailed description of information-system-report.

  See section 4.2.1 for a more detailed description of tracking-report.

- **Notify-advisory-system-action** is a type. Its parent types are action. Notify-advisory-system-action has no local slots. Notify-advisory-system-action inherits 8 other slots from its parents.

- **Remove-repository** is an action of type action-name and operation*.

  Remove-repository is a pf-type.

  The unfilled slots are post, normal-post, normal-effects, normal-pre, pre, and input.

- **Report** is a type. Its parent types are fr. Report has local slots data-in, data-out, and query. Report inherits 1 other slots from its parents.

- **Return-lending-repository** is an action of type action-name and operation*.

  Return-lending-repository is a pf-type.

  The unfilled slots are post, normal-post, normal-effects, normal-pre, pre, and input.

- **State-changes** is a type. Its parent types are data-observed. State-changes has no local slots. State-changes inherits 1 other slots from its parents.

- **Track-target-tracking-system-action** is a type. Its parent types are action. Track-target-tracking-system-action has no local slots. Track-target-tracking-system-action inherits 8 other slots from its parents.

- **Tracking-information-system** is a type. Its parent types are information-system and tracking-system. Tracking-information-system has local slots additional-information, options, and tracked-state. Tracking-information-system inherits 14 other slots from its parents.

- **Tracking-report** (see section 4.2.1) is a type. Its parent types are report. Tracking-report has local slots accessed-information and target. Tracking-report inherits 4 other slots from its parents.

- **Transaction** is a type. Its parent types are operation* and fr. Transaction has local slots user. Transaction inherits 7 other slots from its parents.

- **ULDB** (see section 4) is an advisory-system and tracking-information-system.

  The main purpose of ULDB is to track the state of physical entities. patrons and staff act in the environment causing changes to the state of the UL which are reflected in ULDB. The ULDB attempts to maintain an image of the state of UL by processing requests issued by Users. The ULDB maintains a catalog of records which mirrors the state of the tracked target. These records may be created and destroyed and may be used to generate informational reports.

- **Unshelf** (see section 4.1.4) is an action-tracking-operation.

  Unshelf records the remove-repository of copy-of-book objects from UL by unshelf.the-agent (unknown).

- **Unshelf-all** (see section 4.1.5) is an action-tracking-operation.

  Unshelf-all records the remove-repository of copy-of-book objects from UL by unshelf-all.the-agent (unknown).

- **Users** is an agent.
- **Users-agent-information-system** is a type. Its parent types are people and users-agent.

5 Requirement's State

This chapter summarizes the state of the requirement as of 5/02/90 16:50:49. Generally it points out incompleteness in the document.

5.1 Unindexed Facts

Following are true facts that are not indexed anywhere in the structure of the requirements.

- (∀ (?record) it-is-true (not (and (available ?record ULDB) (*checked-out ?record* ULDB))))
- (inverse remove-repository add-repository)
- (groups (!type book) (!type copy-of-book))
- (inverse check-out check-in)
- (tracks-tracking-system ULDB UL)

5.2 Exceptions

The following premises are false which indicate disproved defaults or invalidated decisions.

- (!default 2 (= (item-states ULDB) (item-states (target ULDB)))) is a disproved default.
- (!default 19 (= (objects unshelf-all) (set-of-uid (unique-id (object-type unshelf-all))))) is a disproved default.
- (user= (data-in last-borrower-of-*book*) (!decl ((?a-*book* *book*)))) is false.
- (!default 15 (five-state-model (item-states UL))) is a disproved default.
- (user= remove (records check-out)) is false.

5.3 *Pending Agenda Items*

Following are pending agenda items.
The following new terms have not been defined: user-target, copy-num, last-borrower, topic, %topic, %title, %author, author, and title.

- *(handled-by (tracks ULDB UL)) is missing* a value
- (handled-by (track-target-state ULDB)) is missing a value
- (handled-by (ensure-accuracy ULDB)) is missing a value
- (handled-by (generate-information-reports ULDB)) is missing a value
- (handled-by (access-limits UL)) is missing a value
- (handled-by rl-remove) is missing a value
- (handled-by (monitor-and-authorize ULDB)) is missing a value
- (handled-by rl-add) is missing a value

- (handled-by (!text-need A Borrower may borrow no more than <Book-Limit> Books at a time.)) is missing a value
- (handled-by (!text-need A Borrower may not borrow two copies of the same book.)) is missing a value
- (handled-by r2) is missing a value
- (handled-by r3) is missing a value

5.4 Missing Values

Following are values which were missing and desired for printing of the preceding document.

- the-agent of unshelf-all
- the-agent of unshelf
- the-agent of check-out
- the-agent of check-in
- the-agent of acquisition
- collection-type of ULDB
- privileged-group of UL.access-limits
- restricted-action of UL.access-limits
- advice-recipients of ULDB
- system-monitored of ULDB

5.5 Outstanding Contradictions

The outstanding contradictions in the requirements are:
There are no outstanding contradictions.

Bibliography

[1] R. Balzer, N. Goldman, and D. Wile. Informality in program specifications. *IEEE Transactions on Software Engineering*, 4(2):94–103, March 1978.

[2] G. Duffy. Categorical disambiguation. In *5th National Conference on Artificial Intelligence*, pages 1079–1082, 1986.

[3] G. Sussman. The virtuous nature of bugs. In *Proc. of Conf. on Artificial Intelligence and the Simulation of Behavior*, July 1974.

[4] J. Schmolze and T. Lipkis. Classification in the KL-ONE knowledge representation system. In *8th International Joint Conference on Artificial Intelligence*, pages 330–332, 1983.

[5] C. Rich and R. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer*, 21(11):10–25, November 1988.

[6] Y. Tan. A program design assistant. MIT/AI/WP 327, MIT, June 1989.

[7] R. Babb et al. Workshop on models and languages for software specification and design. *IEEE Computer*, 18(3):103–108, March 1985.

[8] T. Bell, D. Bixler, and M. Dyer. An extendable approach to computer-aided software requirements engineering. *IEEE Transactions on Software Engineering*, 3(1):49–59, January 1977.

[9] R. Balzer and N. Goldman. Principles of good software specification and their implications for specification languages. In *Conference on Specifications of Reliable Software*, pages 58–67, Boston, MA 1979.

[10] R. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, January 1985.

[11] P. Brown. The interactive system designer's workbench. Lecture at MIT, August 1989.

[12] B. Meyer. On formalism in specifications. *IEEE Software*, pages 6–26, January 1985.

[13] B. Smith. The limits of correctness. *ACM SIGCAS Computers and Society*, 14/15(4/1,2,3):18–26, 1985.

[14] W. Johnson. Specification as formalizing and transforming domain knowledge. In *AAAI-88 Workshop on Automating Software Design*, 1988.

[15] D. Barstow et al. Observations on specifications and automatic programming. In *3rd International Workshop on Software Specification and Design*, pages 89–90, August 1985.

[16] W. Johnson. Deriving specifications from requirements. In *10th International Conference on Software Engineering*, pages 428–438, April 1988.

[17] J. Hagelstein. Declarative approach to information systems requirements. *Knowledge-Based Systems*, 1(4):211–219, September 1988.

[18] IEEE guide to software requirements specification. ANSI/IEEE Std 830-1984, 1984.

[19] C. Potts and A. Finkelstein. Building formal specifications using "structured common sense". In *4th International Workshop on Software Specification and Design*, pages 108–113, April 1987.

[20] C. Potts et al. "Structured common sense:" a requirements elicitation and formalization method for modal action logic. Deliverable Report 2, FOREST, November 1986.

[21] M. Feather and P. London. Implementing specification freedoms. ISI/RR 83-100, USC/ISI, April 1983.

[22] Matsumura et al. An application of structural modeling to software requirements analysis and design. *IEEE Transactions on Software Engineering*, 13(4), April 1987.

[23] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specification. *IEEE Computer*, 18(4):82–91, April 1985.

[24] S. Greenspan. *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition.* PhD thesis, University of Toronto CSRG-155, March 1984.

[25] *Fourth International Workshop on Software Specification and Design.* Computer Society Press of the IEEE, April 1987.

[26] J. Wing. A study of 12 specifications of the library problem. *IEEE Software*, 5(4):66–76, July 1988.

[27] W. Swartout. The Gist behavior explainer. In *3rd National Conference on Artificial Intelligence*, pages 402–407, August 1983.

[28] Paul Lefelhocz. An experiment in knowledge acquisition for software requirements. MIT/AI/WP 330, MIT, 1990.

[29] W. Robinson. Integrating multiple specifications using domain goals. In *5th International Workshop on Software Specification and Design*, pages 219–226, 1989.

[30] W. Robinson. Negotiation behavior during multiple agent specification: A need for automated conflict resolution. CIS/TR 89-13, University of Oregon, September 1989.

[31] S. Fickas. Automating the specification process. CIS/TR 87-05, University of Oregon, December 1987.

[32] MIT Libraries. Welcome to Barton! Informational Sheet, 1988.

[33] C. Rich and R. Waters. Toward a requirements apprentice: On the boundary between informal and formal specifications. MIT/AI/Memo 907, MIT, July 1986.

[34] W. Swartout. Gist English generator. In *2nd National Conference on Artificial Intelligence*, 1982.

[35] C. Rich. The layered architecture of a system for reasoning about programs. In *9th International Joint Conference on Artificial Intelligence*, pages 540–546, 1985.

[36] Y. Feldman and C. Rich. Bread, Frappe, and Cake: The gourmet's guide to automated deduction. In *Proc. 5th Israeli Symp. on Artificial Intelligence*, December 1988.

[37] Y. Feldman and C. Rich. Pattern-directed invocation with changing equations. *Journal of Automated Reasoning*, 1989.

[38] R. Kuper. Dependency-directed localization of software bugs. MIT/AI/TR 1053, MIT, May 1989.

[39] R. Simmons. Combining associational and causal reasoning to solve interpretation and planning problems. MIT/AI/TR 1048, MIT, September 1988.

[40] L. Lamport. *LaTex: A Document Preparation System*. Addison Wesley, 1986.

[41] C. Rich, R. Waters, and H. Reubenstein. Toward a requirements apprentice. In *4th International Workshop on Software Specification and Design*, pages 79–86, April 1987.

[42] H. Reubenstein. A requirements analyst's apprentice: A proposal. MIT/AI/WP 290, MIT, September 1986.

[43] N. Iscoe, J. Browne, and J. Werth. Modeling domain knowledge: An object-oriented approach to program specification and generation. In *submitted to* OOPSLA-89. University of Texas, March 1989.

[44] Selected tools: Programming pearls and case studies from ACM. ACM Press, 1985.

[45] J. Wertheimer. Derivation of an efficient rule system pattern matcher. MIT/AI/TR 1109, MIT, February 1989.

[46] H. Putnam. Meaning holism. Harvard University, MIT 9.671 Class Notes, 1987.

[47] J. Fodor. Semantics Wisconsin style. *Synthese*, 59, 1984.

[48] N. Block. Advertisement for a semantics for psychology. In P. French et al., editor, *Midwest Studies in Philosophy X*.

[49] J. Fodor. Narrow content and meaning holism. MIT 9.671 Class Notes, 1987.

[50] R. Waters. KBEmacs: A step toward the programmer's apprentice. MIT/AI/TR 753, MIT, May 1985.

[51] W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, July 1982.

[52] J. Boose. A survey of knowledge acquisition techniques and tools. *Knowledge Acquisition*, 1(1):3–38, March 1989.

[53] J. Diederich, I. Ruhmann, and M. May. KRITON: A knowledge acquisition tool for expert systems. *International Journal of Man-Machine Studies*, 26(1):29–40, January 1987.

[54] M. Linster. Towards a second generation knowledge acquisition tool. *Knowledge Acquisition*, 1(2):163–184, June 1989.

[55] C. Jacobson and M. Freiling. ASTEK: A multi-paradigm knowledge acquisition tool for complex structured knowledge. *International Journal of Man-Machine Studies*, 29(3):311–328, September 1988.

[56] D. Lenat, M. Prakash, and M. Shepherd. CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *AI Magazine*, Winter:65–85, 1986.

[57] D. Barstow. Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1321–1336, November 1985.

[58] J. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, September 1984.

[59] S. Wrobel. Design goals for sloppy modeling systems. *International Journal of Man-Machine Studies*, 29(4):461–, October 1988.

[60] K. Morik. Acquiring domain models. *International Journal of Man-Machine Studies*, 26(1):93–104, January 1987.

[61] R. Davis. Interactive transfer of expertise: Acquisition of new inference rules. *Artificial Intelligence*, 12:121–157, 1979.

[62] R. Davis. Knowledge acquisition in rule-based systems – knowledge about representations as a basis for system construction and maintenance. In *Pattern-Directed Inference Systems*, pages 99–133. Academic Press, 1978.

[63] K. Ruberg, S. Cornick, and K. James. House Call: Building and maintaining a rule-base. *Knowledge Acquisition*, 1(4):379–401, December 1989.

[64] H. Reubenstein. OPMAN: An OPS5 rule base editing and maintenance package. Master's thesis, MIT, 1985.

[65] L. Eshelman, D. Ehret, J. McDermott, and M. Tan. MOLE: A tenacious knowledge-acquisition tool. *International Journal of Man-Machine Studies*, 26(1):41–54, January 1987.

[66] J. Bennett. ROGET: A knowledge-based system for acquiring the conceptual structure of a diagnostic expert system. *Journal of Automated Reasoning*, 1:49–74, 1985.

[67] J. Hayes and H. Simon. Understanding written problem instructions. In L. Gregg, editor, *Knowledge and Cognition*, chapter 8. J. Wiley and Sons, 1974.

[68] T. Gruber. A method for acquiring strategic knowledge. *Knowledge Acquisition*, 1(3):255–278, September 1989.

[69] S. Marcus. Understanding decision ordering from a piecemeal collection of knowledge. *Knowledge Acquisition*, 1(3):279–298, September 1989.

[70] M. Shaw and B. Gaines. Comparing conceptual structures: Consensus, conflict, correspondence and contrast. *Knowledge Acquisition*, 1(4):341–364, December 1989.

[71] L. Lefkowitz and V. Lesser. Knowledge acquisition as knowledge assimilation. *International Journal of Man-Machine Studies*, 29(2):215–, August 1988.

[72] M. Musen, L. Fagan, D. Combs, and E. Shortliffe. Use of a domain model to drive an interactive knowledge-editing tool. *International Journal of Man-Machine Studies*, 26(1):105–, January 1987.

[73] W. Gale. Knowledge-based knowledge acquisition for a statistical consulting system. *International Journal of Man-Machine Studies*, 26(1):55–64, January 1987.

[74] J. Alexander et al. Ontological analysis: An ongoing experiment. *International Journal of Man-Machine Studies*, 26(4):473–486, April 1987.

[75] J. Boose, D. Shema, and J. Bradsi. Recent progress in AQUINAS: A knowledge acquisition workbench. *Knowledge Acquisition*, 1(2):185–214, June 1989.

[76] K. Handa and S. Ishizaki. Learning importance of concepts: Construction of representative network. *Knowledge Acquisition*, 1(4):365–378, December 1989.

[77] P. Politakis and S. Weiss. Using empirical analysis to refine expert system knowledge bases. *Artificial Intelligence*, 22:23–48, 1984.

[78] K. Silvestro. Using explanations for knowledge-base acquisition. *International Journal of Man-Machine Studies*, 29(2):159–170, August 1988.

[79] N. Haas and G. Hendrix. An approach to acquiring and applying knowledge. In *1st National Conference on Artificial Intelligence*, pages 235–239, 1980.

[80] P. Velardi et al. Acquisition of semantic patterns from a natural corpus of texts. SIGART *Newsletter, special issue on knowledge acquisition*, 108:115–123, April 1989.

[81] C. Green et al. Report on a knowledge-based software assistant. In C. Rich and R. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann, 1986.

[82] S. Lee and S. Sluizer. SXL: An executable specification language. In *4th International Workshop on Software Specification and Design*, pages 231–235. IEEE Computer Society Press, April 1987.

[83] S. Fickas. Automating the analysis process: An example. In *4th International Workshop on Software Specification and Design*, April 1987.

[84] S. Fickas and P. Nagarajan. Being suspicious: Critiquing problem specifications. In *7th National Conference on Artificial Intelligence*, pages 19–24, 1988.

[85] S.Fickas and P. Nagarajan. Critiquing software specifications. *IEEE Software*, pages 37–47, November 1988.

[86] A. Czuchry and D. Harris. KBRA: A new paradigm for requirements engineering. *IEEE Expert*, 3(4):21–35, Winter 1988.

[87] V. Kelly and U. Nonnenmann. Inferring formal software specifications from episodic descriptions. In *6th National Conference on Artificial Intelligence*, pages 127–132, July 1987.

[88] D. Teichroew and E. Hershey. PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems. *IEEE Transactions on Software Engineering*, 3(1):41–48, January 1977.

[89] P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, 8(3):250–269, 1982.

[90] P. Zave. Executable requirements for embedded systems. In *5th Int. Conference on Software Engineering*, March 1981.

[91] J. Guttag, J. Horning, and J. Wing. The Larch family of specification languages. *IEEE Software*, pages 24–36, September 1985.

[92] J. Wing. A Larch specification of the library problem. In *4th International Workshop on Software Specification and Design*, pages 34–41. IEEE Computer Society Press, April 1987.

[93] C. Potts. Requirements analysis, domain knowledge, and design. Technical Report STP-313-88, MCC, February 1989.

[94] C. Potts and G. Bruns. Recording the reasons for design decisions. In *10th International Conference on Software Engineering*, pages 418–427, April 1988.

[95] W. Johnson and M. Feather. Representing evolution transformations. In *IJCAI89 Workshop on Automating Software Design*, pages 125–131, August 1989.

[96] W. Johnson. Domain models for incremental specification acquisition. In *OOPSLA '89, Workshop on Domain Modeling for Software Engineering, workshop notes*, pages 86–91, October 1989.

[97] Using the Wisdm team method to define system requirements. Western Institute of Software Engineering, 1986.

[98] A. Crawford. Joint application design: A new way to design systems. In *Guide Int. Proceedings*. Guide Int. Corporation, 1982.

[99] J. Leite. *Viewpoint Resolution in Requirements Elicitation*. PhD thesis, UC Irvine, 1988.

[100] J. Leite. Viewpoint analysis: A case study. In *5th International Workshop on Software Specification and Design*, 1989.

[101] Special Issue. Requirements engineering environments. *IEEE Computer*, 18(4), April 1985.

[102] D. Ross and K. Schoman. SADT structured analysis for requirements definition. *IEEE Transactions on Software Engineering*, 3(1):6–15, January 1977.

[103] W. Mark. Rule-based inference in large knowledge-bases. In *1st National Conference on Artificial Intelligence*, pages 190–194, 1980.

[104] W. Mark. Representation and inference in the Consul system. In *7th International Joint Conference on Artificial Intelligence*, pages 375–381, 1981.

[105] D. Wilczynski. Knowledge acquisition in the Consul system. In *7th International Joint Conference on Artificial Intelligence*, 1981.

[106] J. Yen, R. Neches, and R. MacGregor. Classification-based programming: A deep integration of frames and rules. ISI/RR 88-213, USC/ISI, April 1989.

[107] P. Devanbu, P. Selfridge, B. Ballard, and R. Brachman. A knowledge-based software information system. In *11th International Joint Conference on Artificial Intelligence*, pages 110–115, 1989.

[108] T. Lipkis. A KL-ONE classifier. In *Proceedings of the 1981 KL-ONE Workshop*, pages 126–143, 1981.

[109] T. Kaczmarek, R. Bates, and G. Robins. Recent developments in nikl. In *5th National Conference on Artificial Intelligence*, pages 978–985, 1986.

[110] W. Mark. Realization. In *Proceedings of the 1981 KL-ONE Workshop*, pages 76–87, 1981.

[111] J. Doyle and R. Patil. Language restrictions, taxonomic classifications, and the utility of representation services. MIT/LCS/TM 387, MIT, May 1989.

[112] R. Hall. Parameterizing a propositional reasoner: an empirical study. *To appear in Journal of Automated Reasoning*, 1990.